Investigations into Implementation of an Iterative Feedback Tuning Algorithm into Microcontroller

Grayson Himunzowa

A dissertation submitted for partial fulfillment of the requirements for the degree of the Master of Science in Electrical Engineering at the University of Cape Town

Declaration

I declare that this dissertation is my own unaided work. It is being submitted for partial fulfillment of the degree of the Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in another university.

Signature of Author……

University of Cape Town
May 2008

Abstract

Implementation of an Iterative Feedback Tuning (IFT) and Myopic Unfalsified Control (MUC) Algorithm into microcontroller is investigated in this dissertation. Motivation in carrying out this research emanates from successful results obtained in application of IFT algorithm to various physical systems since the method was originated in 1995 by Hjalmarsson [4].

The Motorola DSP56F807C microcontroller is selected for use in the investigations due to its matching characteristics with the requirements of IFT algorithm. Speed of program execution, large memory, in-built ADC & DAC and C compiler type are the key parameters qualifying for its usage. The Analog Devices ARM7024 microcontroller was chosen as an alternative to the DSP56F807C where it is not available.

Myopic Unfalsified Control (MUC) is noted to be similar to IFT since it also employs 'myopic' gradient based steepest descent approach to parameter optimization. It is easier to implement in that its algorithm is not as complex as the IFT one, meaning that successful implementation of IFT algorithm in a microcontroller would obviously permit the implementation of MUC into microcontroller as well.

Acknowledgements

I give thanks to God for His providence during the hard times of this dissertation period, my supervisor Professor Martin Braae for his consistent supervision, Mr Attfield (technical officer) for providing technical support in the practical aspect of the project, Mr. A.P. Malichi and Mr S.K. Namukolo (Lecturers at the University of Zambia) for continued encouragement during the period of this thesis, and my wife Rebecca for her support during the period of study for this Master of Science Degree in Electrical Engineering.

Contents

Chapter Four

Chapter Five.

List of Figures

List of Tables

List of Acronyms

ADC- Analog to Digital Converter

BFGS- Broyden- Fletcher- Glodard- Shannon

DAC- Digital to Analog Converter

DSP- Digital Signal Processing

IFT- Iterative Feedback Tuning

ISE- Integral Square Error

ISP- In System Programmer

MIPS- Micro-Instruction per second

MRAC- Model Reference Adaptive Control

MUC- Myopic Unfalsified Control

NAN- Not a Number

PID- Potential- Integral- Differential

PLL- Phase Locked Loop

PWM- Pulse Width Modulator

RAM- Random Access Memory

RTP- Rapid Thermal Processing

SPI- Serial Peripheral Interface

ZN- Ziegler-Nichols

Chapter One

Introduction

Investigation into the feasibility of implementing Iterative Feedback Tuning (IFT) and Myopic Unfalsified Control (MUC) into microcontroller, in this research, originates from successful results obtained by Machaba [3]. He carried out an exploratory investigation into IFT algorithm by applying it to a DC motor and comparing its performance with that of a Model Reference Adaptive Control (MRAC) algorithm and showed that the controller in an IFT algorithm in a closed loop was capable of updating the controller parameters with respect to variations in process dynamics with 10% less error [3]. The reason for carrying out this research was to set a base case in which other related adaptive techniques can be studied and compared to find out the most effective one. It was surmised that this could lead to commercialization of IFT or any of the adaptive techniques existing currently.

1.1 Statement of the Problem

Since the IFT algorithm has attracted a lot of interest, in that the results of it from both scientific and industrial world are successful [8], there is a need then to develop or engineer stand-alone hardware that implements the IFT algorithm. Then comparisons can be made between its performance and that of MUC and other adaptive techniques. Research indicates that there is currently no stand-alone IFT algorithm hardware or a commercial product. This is one of the motivations for the present investigations into coming up with such a hardware. In addition, such hardware would provide a platform on which other existing techniques would be implemented.

A particular case of industrial interest is tuning PI or PID controller since classical approaches contain a number of fundamental problems, such as:

(1) The amount of offline tuning required,

(2) The assumption on the plant structure,

(3) The issue of system stability, and

(4) The difficulties in dealing with nonlinear, large time delayed and time variant plant.

To overcome these problems, it has been proposed that the desired closed loop models be free, that is to say they should not be fixed but be able to adjust together with the controller parameters. Hence, abandoning off-line tuning and instead focusing on the problem of controller tuning on-line or automatically. One school of thought addresses these problems as parameter optimization to be carried directly on the controller parameters so as to directly minimize a closed-loop control performance criterion. This applies to IFT and MUC and successful implementation of the duo would mitigate problems highlighted above in industrial control [23].

1.2 Research Objectives

The objectives of this thesis project are:

(1) To investigate the feasibility of implementing both an IFT and a MUC algorithms on a microcontroller. The microcontroller to be used in the research should meet the basic requirements of the two algorithms: large memory (both program flash and RAM); speed of processor (to handle complex mathematical equations that feature in the two algorithms); large stack area (for storage of local variables since IFT should loop continuously); DSP functions (for ease of coding since the algorithms feature complex mathematical expressions).

(2) To carry out an investigation of implementing an IFT algorithm on the microcontroller meeting the characteristics of the two algorithms highlighted above.

(3) To carry out an investigation on Myopic Unfalsified Control (MUC) and compare it with IFT algorithm

1.3 Methodology

{1) Investigations of an IFT algorithm used by other researchers [4, 15, and 23] were undertaken in order to understand the algorithm in readiness for implementation of the algorithm into a microcontroller.

(2) The first stage of this research project was to select a suitable microcontroller that could accommodate and handle the IFT algorithm with features that require large stack area, large memory and fast speed of processing.

(3) Before the commencement of investigations into the feasibility of implementing IFT and MUC algorithms into a microcontroller, a literature review on IFT and MUC theory, applications, trends and developments was undertaken. Intensive literature survey on microcontrollers was also carried out as preparation for coding continuous looping algorithm.

(4) The IFT algorithm was coded in C- language a compiler used in Motorola DSP56F807C microcontroller.

(5) Investigation into IFT and MUC algorithms implemented on a microcontroller was conducted to discover its capability running on a microcontroller.

(6) The results obtained were analyzed and compared with others.

(7) Conclusion and recommendations were made in line with the obtained results.

1.4 Historical Background

Iterative Feedback Tuning is a new method from 1994 [3] but the idea of it has been there since 1958 since its concept originates from Model Reference Adaptive Control (MRAC) [2, 3]. Whitaker and his team originally proposed the Model Reference Adaptive Control in 1958 and further work was done on the method in the sixties and seventies [2]. Whitaker introduced two ideas on the method [2], which are:

(1) A model specifies the performance of the controlled system,

(2) The parameters of the regulator are adjusted based on the error between the outputs from a reference model and the system.

The above two concepts implies that MRAC controller can adapt or modify its behaviour in response to changes in the dynamics of the process and the disturbances. This implies that MRAC is capable of working well over a wide range of operating conditions as compared to ordinary fixed parameter, like linear feedback control which can work well around one operating condition.

Initially MRAC was developed for servo problems in deterministic continuous time systems. Later the ideas and the theory were extended in order to cover discrete time systems with stochastic disturbances [2] which led to the development of the IFT algorithm.

IFT as compared to MRAC is more of a controller parameter optimization problem than model design problem [4]. The idea of improving the performance of on-line, operating controllers, on the basis of closed loop data corresponds to a natural way of thinking [4] that implies that IFT adds intelligent characteristics to controllers. This has always been a dream of the control industry [29].

The IFT was developed to solve many of the problems experienced in industrial control, mainly concerning model identification by utilizing only the input and output data to design a controller on-line for a given plant or process [8]. It was initially derived by Hjalmarsson [4] in an effort to understand the convergence/divergence properties of the iterative identification and control design scheme [15] and has quickly proved its efficiency in both laboratory and industrial applications such as in process control, often for disturbance rejection [16] as will be shown later [8]. As such, IFT schemes are capable of becoming popular in industry for automated optimization of industrial control loops or systems. Hence investigating its implementation into microcontroller is justified as it may lead to commercialization of IFT (and any other adaptive techniques, like MUC).

However, an IFT variant or new IFT criterion was proposed recently in order to solve the problem of non-minimum phase pole-zero cancellation [17]. This is done by letting the choice of the desired model be more free, based on the fact that the IFT method assumes that one has no knowledge or only partial knowledge of the plant and, therefore, that one cannot know in advance whether a certain reference model is achievable (even approximately) or not [17]. Hence, there must be a parameter in the reference model to place zeros otherwise the controller achieves an unstable pole-zero cancellation in order to reduce the phase lag. This is particularly crucial when the process is non-minimum phase.

Another approach known as Unfalsified Control (MUC) that depends on the performance goals and the candidate controller parameterization, was proposed by Tsao and Safonov [22] but due to the computational burden of this technique, a new simplified gradient based version of unfalsified control known as Myopic unfalsified Control was proposed by Jun and Safonov [23]. Unlike the usual global approach, it can only examine the local gradient of the performance function at each time. This is an adaptive control theory that permits learning by an on-line process of elimination of candidate controllers as these are falsified by evolving open-loop plant data. Its goal is to look for inconsistencies between the constraints of the given performance specification that each candidate controller would introduce if inserted in the feedback loop [21].

As stated above, MUC and IFT are similar in some sense. Both are model-free, data-driven methods with similar control criterion. Both employ a “myopic” gradient-based steepest descent approach to parameter optimization.

This project is based on establishing the viability of implementing an IFT algorithm on microcontroller hardware and on comparing it with another technique known as MUC.

1.5 Plan of Development

Chapter two discusses literature review on Iterative Feedback Tuning, Myopic Unfalsified Control and microcontrollers. Chapter three reviews the basic theory of IFT and MUC algorithms. Chapter four describes coding of IFT and MUC Algorithms in C computer language. Chapter Five outlines testing of IFT and MUC Algorithm Code Implemented into a Microcontroller. Chapter Six considers an application of IFT and MUC Algorithm to a DC Motor.

Discussions and conclusions are given in Chapter Seven.

Chapter Two

Literature Survey

Sources of literature in this research were the University of Cape Town library that was used mainly for books, dissertations and journals and the Google Scholar search engine that was used for researching literature electronically mainly for useful web pages, papers and electronic journals.

2.1 Search Objective

Since the objective of the thesis was to investigate the viability of implementing Iterative Feedback Tuning Algorithm or Myopic Unfalsified Algorithm on microcontroller hardware, the initial task was to carry out an intensive literature survey of IFT and MUC theory and their applications. This formed a major part of the project and in addition a great deal of literature on microcontrollers was reviewed and is presented in this chapter.

2.2 IFT and MUC Concept

Iterative Feedback Theory (IFT) like Myopic Unfalsified Control (MUC) is a data-based method for controller optimization [4]. The concept of IFT is to use the repetitive nature of a process to progressively enhance its set point tracking performance. Using error measurements in a previous cycle, the controller parameters are updated iteratively after each set of cycles. These types of controller are able to deal with dynamic systems with imperfect knowledge of dynamic structures and/ or parameters operating repetitively over a fixed time interval [1]. The objective of the IFT is to minimize a quadratic performance criterion. The gradient of the cost function at each step is estimated from data. These data are collected with the actual controller in the loop [16] and are used to optimize the controller.

Recently, as already stated, a variant of IFT algorithm based on a new criterion has emerged mainly to address the problem of pole-zero cancellation in non-minimum phase process plants [17]. Hence, the new IFT criterion has parameters in a reference model that are updated together with controller parameters. In this way, the data obtained during the iterative procedure are used to tune the reference model towards one that is compatible with the plant [17].

In summary IFT minimizes quadratic criterion, J with respect to controller parameters and placement of poles in reference model.

MUC like IFT works on the principle of adjusting controller parameters whenever a controller is falsified. This is done in relationship with measured data by the steepest-descent direction of negative gradient of the cost function ($-\nabla \tilde{\tilde{J}}(\theta,t)$) to meet the requirements of performance specification ($\tilde{J}(i,t) \leq 0$**)**. The detailed explanation is given in Chapter Three and Appendix One.

2.3 IFT and MUC Applications

A vast number of IFT and MUC applications currently existing or described in literature are a motivating factor for investigations into the feasibility of implementing IFT and MUC algorithms on a microcontroller as this could lead to the development of a product for industrial use. Some of these applications are outlined below:

(1) In [5] the algorithm was shown to be very successful in controlling the mass-spring system's position under heavy friction where a two degree of freedom IFT controller was applied to a servo system. Two strategies were adopted in order to deal with this heavy friction: The one was to separate the tuning of the feedback and feed forward controllers and the other was to employ the Broyden-Fletcher-Glodfard-Shanno (BFGS) method as a quasi-Newton method in a parameter update law.

(2) In [6] the IFT algorithm was applied to tune controller parameters based on correlation approach. This was done by making the output error between the desired and the achieved output response uncorrelated with the set point signal. By doing this the IFT scheme can be used for controller tuning.

(3) In [7] the data-driven model free control design method that was introduced by Hjalmarsson in 1994 was extended to a case where both the plant and the controller are allowed to be nonlinear. In this paper it was shown that one can obtain an estimate of the model of the plant experimentally by using the closed loop measured data (input and output signals).

(4) In [8] the IFT method proved to achieve a fast response to set point changes, faster settling time as well as less overshoot compared with other classical PID tuning method namely: The Ziegler-Nichols (ZN) tuning rules, the Internal Model Control (IMC) and the Integral Square Error (ISE) method.

(5) In [9] the Iterative Feedback Tuning algorithm was applied to internal model control (IMC) and the Smith predictor. In this case the algorithm was altered by doing four experiments instead of three to accommodate the tuning of the Smith predictor.

(6) In [10] the relay auto-tuning of the PID controllers using IFT was applied to a process control problem in which the PID controller was auto-tuned to give specific bandwidth and phase margin using an IFT scheme. The algorithm was tested in the laboratory on a coupled tank and the theoretical results were demonstrated to be observed in practice.

(7) In [11] the paper introduces a model free tuning algorithm based on frequency domain properties of the closed loop system signals. The scheme used for this paper exploited the feature of the Iterative Feedback Tuning method.

(8) In [16] the asymptotic convergence rate of IFT for disturbance rejection was analyzed and was found that the convergence rate depended on the covariance of the gradient estimates, The results obtained were used to derive optimal choices for the pre-filter in two different situations like: (1) the current controller is near the optimal controller and (2) a pre-filter which optimally increases the asymptotic accuracy of IFT under a constraint on the energy used during the special feedback experiment was derived and the pre-filter is optimized for accuracy of a single IFT step under the same energy constraint.

(9) In [12] the iterative identification scheme was applied to a sugar cane crushing mill. This paper examined the Zang scheme iterative identification and controller design as applied in a modified form to a sugar cane crushing mill. The selection of excitation and data filtering for model identification is done by using connections between identification and robust control design.

(10) In [13] the Iterative Feedback Tuning was used to tune controllers for linear time invariant (LTI) multivariable systems. The algorithm was applied to a laboratory model of a helicopter.

(11) In [4] Hjalmmarsson presents one the first publications of the iterative feedback tuning method in 1994. It has since been applied to a variety of tasks from the simplest like optimal tuning of simple PID controllers [8] to more complex tasks like systematic design of controllers of increasing complexity.

(12) In [3] it was noted that application to a DC motor had not been previously considered in detail. Since the DC motor is an important electrical engineering device and has dynamics that are very much dependent on its loading, it was chosen as the system in which to evaluate IFT method. These results would give a good indication of the potential of the IFT for engineering applications in mineral extraction industry. In the same work IFT algorithm implemented in visual basic language was used to control a

DC motor and the results were favourable when compared to model reference adaptive control (MRAC) algorithm.

(13) In [24] MUC algorithm was used for real-time PID controller parameter tuning and adaption moving away from traditional methods such as Ziegler-Nicholas method.

(14) In [25] Unfalsified Control was used in detection of wheel and work piece contact/release in reciprocation surface grinding by processing the output of a piezoelectric force transducer. It efficiently detected these events independent of work piece material or grinding conditions (except for dry grinding with low material removal rate). The purpose of this application was to prevent force overshoot at the beginning of engagement in the study of substance plastic deformation.

(15) In [26] Unfalsified Control was implemented on an industrial weigh belt feeder as a means of using both open and closed loop test data to identify a subset of controllers (from an initial set) that met the multiple objectives specified by the control engineer. A novel feature of the unfalsified approach was that it allows the performance of a given controller to be predicted without inserting the controller in the loop. In addition, this methodology does not require an explicit model of the system to be controlled. When the unfalsified PI auto tuning approach was applied to the industrial weigh belt feeder, it was able to successfully identify a subset of PI control laws that meets the performance specification. This automated PI tuning approach was implemented here off-line but may be implemented on-line and applied in a straight forward manner to other control systems.

(16) In [27] Unfalsified control was used to implement control of a Semiconductor Automated Manufacturing process. This novel approach in integrated circuit manufacturing demands fast tracking control laws that achieve near uniform spatial temperature distributions. In order to ensure the final product quality, it was essential to maintain a uniform

temperature profile despite uncertainties in both transient and steady state phases of the process. Specific accomplishments included the development of mathematical and computational tools for heat transfer modeling, specifically conduction and multiband radiation, nonlinear model reduction, methods for robust thermal control, and an approach applicable to repetitive run-to-run feed forward learning control. All the results were tested for feasibility on commercial RTP chambers.

All these applications indicate the wide usage of IFT and MUC techniques but no mention of IFT or MUC microcontroller hardware development was made so far. This becomes a clear indicator that there is a need to carry out the research towards the development of IFT or MUC stand-alone hardware that may lead to commercialization of IFT or MUC and other related techniques. Hence investigations into the feasibility of implementing IFT and MUC on a microcontroller are justified.

2.4 Microcontroller Hardware

In discussing this topic, IFT will be referred to more than MUC since motivation in carrying out this project was as a result of successful results achieved from an IFT work in [3]. Since IFT and MUC have relatively complex algorithms the main task was to identify suitable hardware that could accommodate the algorithms. As given by the title of the project, digital hardware was preferred to analog hardware in that it is impractical to implement such a complex algorithm into an analog hardware. In choosing the hardware the following factors were considered to be critical in ensuring the requirement of IFT algorithm would be met:

(1) Large memory since the IFT visual basic program in [3] had fifteen arrays for data storage and the length of each array was one thousand (1000) elements. These arrays were declared as floats taking four bytes of

memory space per element in the array hence a microcontroller with small memory would fail to run the IFT algorithm,

(2) high speed of processing since IFT algorithm executes in stages that occupy N-length time periods to collect and process data. Data is first collected and stored in an array for N-length time periods before being passed to the next stage for optimization for N-length time periods. In the case of the IFT visual basic program N was made one thousand (1000) taking considerable amount of time. Since the IFT controller sampling time was 50ms in [3] the total required time to complete one experiment was 50s and making it 100s for two experiments.

(3) a suitable compiler such as C that is capable of handling floating points was preferred to other programming languages,

(4) on board Analog to Digital Converter and Digital to Analog Converter were required to create a hardware feedback loop of an IFT. The output of the motor is fed back to the input of the ADC and the input of the motor is fed by the DAC as will be indicated later.

Such functionality is scarce in common microcontroller hardware. Hence a careful search for the right hardware with features as shown above was necessary. The technique used to decide on these features given above for selecting the microcontroller hardware for use in implementation of the IFT was derived from Visual Basic Program used in a previous IFT project [3]. This also included the speed at which IFT ran without causing any problems. This estimate was arrived at by the number of arrays used in the original IFT algorithm coded in Visual basic in [3]. There were fifteen arrays declared as float variables that take four bytes of memory space each. Considering one thousand iterations required in the original algorithm, means the total memory required was sixty thousand (60000), four-byte words for all fifteen (15) arrays.

In line with the characteristic requirement of a microcontroller for use in this research, derived from above calculation, a survey was conducted and a summary of microcontrollers selected are shown in table1.

Table1 Microcontroller Properties

NO.	Microcontroller	Manufacturer	Processor Core Features				
			Speed	Float Point	Word Size	Memory	Peripheral ccts
1.	ARM7024	ETT. CO., LTD (Processor from Analog devices)	40 MIPs	No	16/32 bit	64K flash 8K RAM	ADC; DAC etc.
2.	ADCU832	ETT. CO., LTD (Processor from Analog devices)	17 MIPs	No	16/32 bit	128K flash 16K RAM	ADC; DAC etc.
3.	ET-ARM Stamp	ETT. CO., LTD (Processor from Philips)	16 MIPs	No	16/32 bit	12K flash 0.5K RAM	ADC etc.
4.	ET-AVR Stamp	Atmel	40 MIPs	No	16/32 bit	128K flash 4K RAM	ADC etc.
5.	DSP56F807C	Freescale Semiconductor	40 MIPs	No	16 bit	64K flash 6K RAM	ADC etc.

It must be noted that table1 indicates only necessary processor features that are required for implementation of the IFT and MUC algorithms. Detailed information can be sought from data sheets of the respective microcontrollers given above.

There are many microcontrollers available on the market such as 8051 core microcontrollers, MCU microcontrollers, DSP56800 microcontrollers etc.

The Motorola DSP56F807 microcontroller was chosen for use in this research mainly due to its good characteristics as outlined below and summarized in table1. The DSP56F807C is made of DSP56800 core. This architecture with 16-bit multiple-bus processor is designed for efficient real-time digital signal processing and general purpose computing [14] hence makes it easy to code advanced

algorithm such as IFT or MUC. It is composed of functional units that operate in parallel to increase the throughput of the machine hence decreasing the execution time of each instruction. For example it is possible for the data arithmetic logic unit (ALU) to perform a multiplication in a first instruction, for address generation unit (AGU) to generate up to two addresses for a second instruction, and for a program controller to be fetching a third instruction. This feature, parallel operation, normally does not exist in non DSP microcontrollers. Comparing the DSP56F807C chip to the ARM7024 hardware manufactured by Analog Devices, it was discovered that they had similar characteristics as from table1. The only difference was architecture.

The advantage of using microcontrollers, as opposed to using larger microprocessors, is that the parts-count and design costs can be kept to a minimum. Justification is given later in the definition of a microcontroller in Appendix Two. Applications with microcontrollers can easily be customized to suit desired applications, for example a dedicated controller can be developed for specialized applications where optimal investment is required instead of using a personal computer that is costly and meant for general-purpose applications.

Research has revealed that microcontrollers are already in use in implementing proportional-integral-differential (PID) controllers. This observation is based on the number of hits (19500 hits) obtained from the Google scholar search engine for the term 'PID' or 'PID and implement'. There were 87 hits for microcontroller based Model Reference Adaptive Control and none for IFT. It is still not known whether a microcontroller implementation of the IFT algorithm does exist indicating that IFT is a new technique still in active research and has not been commercialized yet. Hence, implementation of an IFT algorithm into microcontroller if achieved could lead to the development of an economical and optimal controller. This is an important objective for industrial control and will form a base-case against which other similar techniques could be compared.

Important features of DSP56F807C microcontroller are briefly highlighted and related to the requirement of IFT algorithm:

(1) A speed of 40 MIPS at 80MHz frequency was suitable for execution of control algorithms like IFT that runs in two stages (coded as procedures 'experiment1' and procedure 'experiment2') for the case of one degree of freedom systems. In procedure 'experiment1' data points are collected from a process plant and these data points can sometimes amount to one thousand data points as shown in [3];. Speed requirement becomes crucial in higher degree of freedom systems because the number of experiments increases making it difficult to implement the algorithm into microcontroller hardware. C language allows breaking the algorithm into segments that can be called experiments.

(2) 2K by 16-bit words of program RAM allows the use of interrupt vectors that are needed for internal and external interrupts. For example the operation of analogue-to-digital converter requires interrupt service request.

(3) 8K by 16-bit words of data flash was required for the usage of dynamically linked libraries (DLLs). Since the experiments of IFT algorithm possess a large number of local variables (due to a large number of arrays) where stack cannot store them the only option could be the usage of dynamic memory.

(4) 4K by 16-bit words data RAM for usage of stack and other related processor registers. When the RAM is large it means that stack can store a large amount of temporary data and this was a requirement in the IFT algorithm since it was designed to take about six separate files of C- code, namely: main body, experiment1, experiment2, ADC, DAC, and signal files. The local variables of the subroutine programs can be kept on the stack as main calls respective subroutines.

(5) 128K data and program memory expansion capabilities are necessary feature for trying more sophisticated features of IFT and MUC such as second or third degree of freedom.

(6) Four 12-bit, Analog-to–Digital Converters used as transducer for converting analogue signal (y) from the output of a process plant to a digital signal within the controller.

The ADC is in-built in the DSP56F807C microcontroller while the DAC is an external device interfaced to the microcontroller through the SPI peripheral component. With combination of ADC and DAC on DSP56F807C microcontroller makes it easier the implementation of IFT or MUC algorithm as shown in figure2.1. IFT algorithm resides into the microcontroller memory as shown in the figure below.

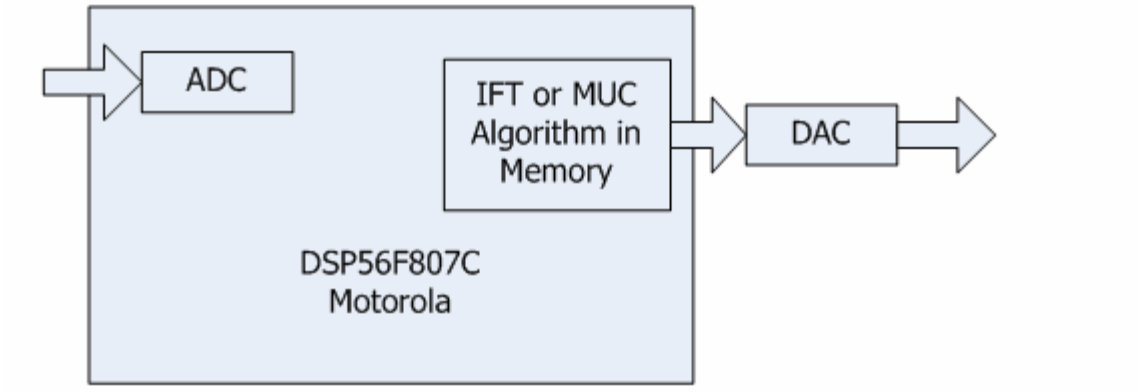

Figure 2.1 DSP56F807C microcontroller with IFT or MUC Algorithm embedded

Chapter Three

Iterative Feedback Tuning and Myopic Unfalsified Control Theory

In this chapter the basic concept of Iterative feedback Tuning (IFT) and Myopic Unfalsified Control (MUC) are reviewed in order to investigate the possibility of implementing these control laws on the constrained hardware provided by the DSP56F807C microcontroller. MUC and IFT are similar in the terms of their algorithms and also both are model-free, data driven methods [21]. Hence if it is feasible to implement IFT onto the DSP56F807C microcontroller then it would be feasible to implement MUC on to the DSP56F807C microcontroller as well.

Single-Input-Single-Output IFT was adopted in this research due to the limitations in memory that is in-built in the DSP56F807C microcontroller. As stated in Chapter One, IFT is a method for tuning parameterized controller in a feedback loop when a mathematical description of a plant is not available and the controller must be tuned on the basis of input-output measurements. In IFT, tuning of the controller parameters is performed through an iterative procedure where a sequence of parameter updates is calculated and implemented to check the performance until there is convergence in parameter update. While in MUC, input and output data are measured and compared with performance specification to falsify or unfalsify that controller. The detailed mathematical derivations are given in Appendix One [3, 4, and 15].

3.1 The Idea of Iterative feedback Theory

The concept of the iterative feedback tuning algorithm is illustrated in the block diagram in figure 3.1 [3]. The controller k is type 1 (and represents a one degree of freedom transfer function), g is an unknown model representing the plant and m is the desired model.

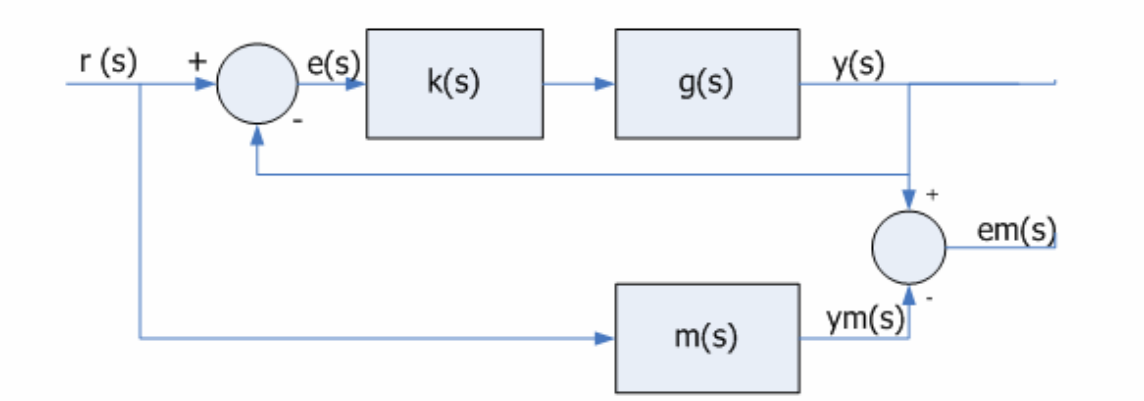

Figure 3.1 Block diagram of a one degree of freedom controller in closed loop system.

1) The Dynamic Models

The equations governing the closed loop dynamics (assuming the noise and disturbances are zero) are

$$y=Tr \tag{3.1}$$

where T is the transfer function of the closed loop system.

Equations for desired model for open loop system yields:

$$y_m = mr \tag{3.2}$$

$$e_m = y - y_m \tag{3.3}$$

The modeling error, e_m is collected in an array by procedure 'experiment1' for N-length time and eventually passed over to procedure 'experiment2' for optimization. Hence, storage facility for e_m was a 'float-precision' declared array requiring memory space of 4 bytes * N. For the case of IFT Visual Basic program, 4kb was required for this array since N was made one thousand. When summing e_m as

shown in (3.4) the result was packed in an array also occupying 4kb of memory space.

(2) The Cost Function

The cost function is a scalar based and is given by

$$\mathbf{J} = \frac{1}{2}\sum_{1}^{N} e_m^2(t) \tag{3.4}$$

as described in Appendix One.

The model for a controller, k in figure 3.1 is given as

$$\text{k (z)} = \frac{\rho_1 z + \rho_0}{(z-1)} \tag{3.5}$$

This was chosen for simplicity sake, and to save memory as this is one of the key constraints assumed for the present research project. Secondly, the proportional-plus-Integral (PI) controller is commonly used in industry.

The gradient of cost function criterion is represented by:

$$\frac{\partial J}{\partial \rho} = -\sum_{1}^{N} e_m * \frac{\partial e_m}{\partial \rho} \tag{3.6}$$

A detailed explanation on the gradient of the cost function in terms of controller parameter vector is given in Appendix One. The memory requirement for equation (3.6) was reduced to eight bytes as compared to 8kb in the visual basic program since there was no need to make it an array.

As derived in Appendix One the required gradient, from equation (3.6) becomes

$$\frac{\partial y}{\partial \rho} = \frac{1}{k} * \left[T * r - T^2 * r\right] \tag{3.7}$$

This gradient is used in IFT method to adjust or tune the controller parameters.

In this application the controller parameters form a vector and the update of controller parameters is given by the following equation:

$$\rho_{i+1} = \rho_j - \gamma_j R_j^{-1} \frac{\partial J}{\partial \rho}(\rho_j) \tag{3.8}$$

Updated parameters were kept in variable type 'float' that takes 8 bytes of memory space meaning 8 bytes for two parameters.

(3) Input signals for the one degree of freedom controller

From the IFT control loop shown in figure 3.1 the process input is obtained from the controller as shown in the transfer function equation:

$$u = k*(r-y) \tag{3.9}$$

The gradient of the input with respect to the controller parameter vector is given by:

$$\frac{\partial u}{\partial \rho} = \frac{1}{k} * \frac{\partial k}{\partial \rho} * u^{(2)} \tag{3.10}$$

The input gradient is used together with output gradient to achieve controller parameter optimization. The superscript (2) indicates the control signal, u existing in procedure 'experiment2'.

3.2 Myopic Unfalsified Control Theory (MUC)

As stated above, MUC and IFT are related in some sense. Both employ a 'myopic' gradient-based steepest descent approach to parameter optimization. MUC is simpler and easier to implement as compared to IFT [21] and for this reason it was also investigated as to whether it can run on to the DSP56F807C microcontroller.

The concept of MUC algorithm is illustrated in the block diagram in figure 3.2. As before the controller k is type 1 (and a set pre-defined controllers) controller, and g is unknown model representing the plant. Note that, unlike the IFT of figure 3.1, there is no desired model.

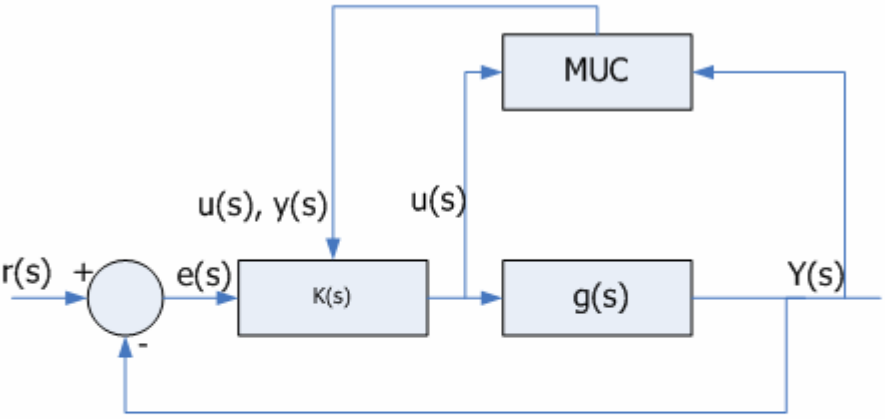

Figure 3.2 Block diagram of a MUC controller in closed loop system.

A MUC controller is said to be falsified by measurement information if this information is sufficient to deduce that performance specification (r, y, u) $\in$ Tspec $\forall$ r$\in$ R would be violated if that controller were in feedback loop. Otherwise, the controller is said to be unfalsified [21].

Unfalsified Control method has only three elements:

(1) Data is required for observation. It is data that assess whether performance objective is not met and then the controller is tested for falsification.

(2) Candidate controller hypotheses are set of controllers that are tested for falsification or unfalsification. If the controller is falsified it is discarded. In case of Myopic Unfalsified Control, the controller is not discarded but

adjusted in the steepest-descent direction $-\nabla \tilde{\tilde{J}}(\theta,t)$ so that the performance specification, $\tilde{\tilde{J}}(\theta,t)$ tends to decrease whenever the currently active controller parameter vector θ is falsified

(3) Goals may be described as control laws or likened to desired models in IFT. Data are observed to be consistency with goals and if there is any discrepancy then particular controller becomes falsified.

These elements are sieved by the computer so that controllers that falsify data are removed and only those that do not falsify data are kept.

3.3 Myopic Unfalsified Control Algorithm

(1) Measure input, u(t) and output, y(t) from an unknown plant

(2) Use u(t) and y(t) to calculate the cost function, $\tilde{\tilde{J}}(\theta,t)$ thus

$$\tilde{J}(\theta,t) = -\rho + \int_0^t T_{spec}(r(\tau),u(\tau),y(\tau))d\tau \qquad 3.11$$

$\rho \geq 0$ and $T_{spec}(.,.,.)$ are chosen by the designer.

(3) A candidate controller ki becomes unfalsified at time τ by plant data u (t), y (t) if, and only if $\tilde{J}(i,t) \leq 0$.

(4) If the controller ki is falsified then equation 3.11 becomes

$$\nabla \tilde{J}(\theta,t) = \int_0^t T_{spec}(\tilde{r}(\theta,\tau),u(\theta,\tau),y(\theta,\tau).\nabla \tilde{r}(\theta,\tau)d\tau \qquad 3.12$$

$T_{spec}(\tilde{r}(\theta,\tau),u(\theta,\tau),y(\theta,\tau)$ =

$$\left|w_{1*}(r(\tau)-y(\tau))\right|^2 + \left|w_2 * u(\tau)\right|^2 - \sigma^2 - \left|r(t)\right|^2$$

$$\nabla\tilde{r}(\theta,\tau) = -K(\theta)^{-1}\nabla K(\theta)K(\theta)^{-1}u(\tau)$$

$$\nabla K(\theta) = \left[\frac{\partial K(\theta)}{\partial \theta_1} \frac{\partial K(\theta)}{\partial \theta_2} \cdots \frac{\partial K(\theta)}{\partial \theta_n} \right]^T$$

Operator '*' denotes convolution operation and w_1 and w_2 are filters. In the MUC algorithm implemented onto microcontroller, filters, w_1 and w_2 were ignored.

Therefore controller parameter adaptation equation can be expressed as:

$$\dot{\theta} = \gamma \int_0^t \frac{\partial T_{spec}(\tilde{r}, \tau)}{\partial \tilde{r}} K(\theta)^{-1} \nabla K(\theta) K(\theta)^{-1} u(\tau) d\tau$$

Where γ **>** 0 is a design constant that determines the rate of adaptation

(5) If and only if $\tilde{J}(i,t) \leq 0$, then parameter updates otherwise does not update.

3.4 Memory Requirements for MUC

From the algorithm above variables u (t), y (t) and $\tilde{J}(\theta,t)$ were declared as floats taking a memory space of 24 bytes. This means that for one thousand (1000) data points, memory required by the three variables would still be 24bytes which is very minimal as compared to the memory requirements of the IFT algorithm. However, with reference to chapter two, DSP56F807C and ARM7024 microcontrollers both have sufficient memory capacity (64kb and 128kb respectively) that is capable of handling such a memory requirements.

Chapter Four

Coding IFT and MUC Algorithms.

In this chapter the IFT and the MUC algorithm codes are being formulated, first by use of flowcharts and later the actual C-language for microcontrollers. Flowcharts are presented within the chapter while the C code is in Appendix Three and will always be referred where necessary.

4.1Coding IFT Algorithm

The equations yielded by the IFT theory as discussed in Chapter Three were coded in the Motorola DSP56F807C and ARM7024 C-language. The details are provided in this chapter. An overview of the IFT program is given in tables AP4.1 and AP4.2 of Appendix Four and was translated into C-language for the DSP56F807 and ARM7024 microcontroller. The structure of the program is depicted in figure 4.1.

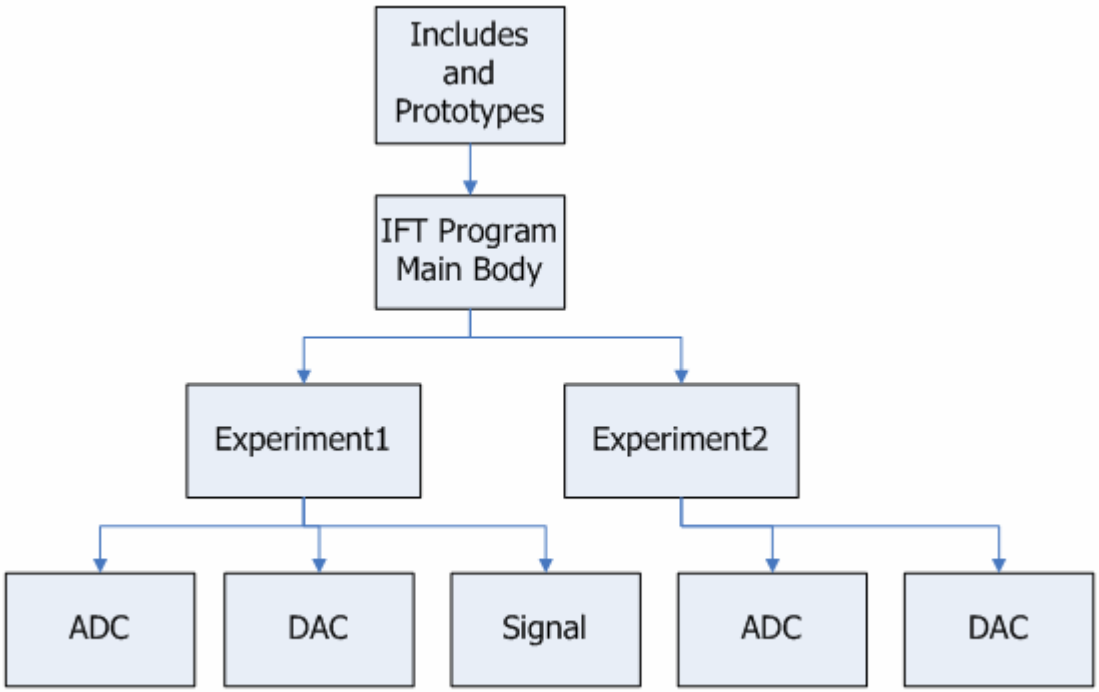

Figure 4.1 IFT Program Structure.

The blocks of the program structure is what constitutes IFT algorithm coded in C-language. Brief descriptions of each block in this figure are provided following a top down order.

(1) Includes and Prototypes

'Includes' are collection of functions and variables, in the form of libraries, that are common to the standard C++ implementation and normally kept in header files [19].

In the 'Includes and Prototypes' block, libraries containing vast number of functions and variables common to standard C++ implementations were included to avoid repetitions in defining such variables and functions in different sections of the IFT and MUC programs. This ensures that every function except the 'main function' was declared before being used. The declaration of the function tells

The compiler how the function will be used: In particular, what kinds of arguments it takes and what value, if any it returns [19]. By use of the directive command '#include', standard input and output functions like 'stdio library' were declared. The benefit of including 'stdio library' is to avoid re-coding common functions such as printf and scanf needed for printing out and inputting data for the set point signal to procedure 'experiment2' and output the signal from function 'experiment2'.

Another 'include' required in the implementation of IFT or MUC algorithm was a mathematical library for mathematical functions. Operations carried out on expressions such as the equations for the quadratic criterion, J for IFT or MUC algorithm are functions from mathematical library kept in a header file known as the 'math.h' header file.

The IFT algorithm functions written for this project are listed below:

(1) Experiment1()

(2) Experiment2()

(3) Signal()

(4) ADC()

(5) DAC()

These were declared as function prototypes in the same block to provide type information to the compiler so that it knows what type of arguments to expect. All the functions above were declared void since they do not need to return any value.

In the implementation of the IFT algorithm some includes were automatically generated by the compiler during the compilation period. These are procedural and shared modules which are used for the whole project.

(2) Program Main Body

This is the main function or special function of the IFT algorithm. The program begins executing at the beginning of this function. The body of the main program is shown by a flowchart in figure 4.2. The main program calls other functions, imbedded in it as needed. The imbedded functions in main are:

(1) Experiment1() which in turn calls Signal(), ADC(), and DAC().

(2) Experiment2 which in turn calls ADC() and DAC()

Procedure 'experiment1' generates a set point signal, eta and modeling error signal, emta for procedure 'experiment2' as this is required as data for processing or optimizing controller parameters. These two signals are stored in arrays since each experiment takes N-length time before the program control is passed over to the other function. Communication between procedure 'experiment1' and procedure 'experiment2' is provided the through the 'main body' function.

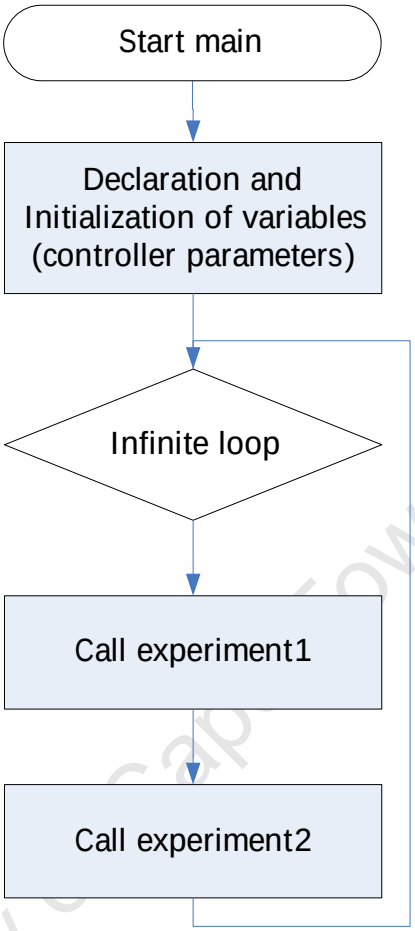

Figure 4.2 IFT Program Main Body Flowchart

The first block in the flowchart initializes controller parameters, ρ_0 and ρ_1. This is followed by a 'for loop' configured as infinite loop to make main function loop continuous. Inside the infinite loop are imbedded functions that call subroutine programs (experiment1 and experiment2) repeatedly as long as the algorithm is in use.

(3) Procedure 'Experiment1'

As already highlighted in section 4.1 item number two, the function 'experiment1' prepares information for the function 'experiment2' in order to carry out the optimization of controller parameters. The flowchart is depicted in figure 4.3.

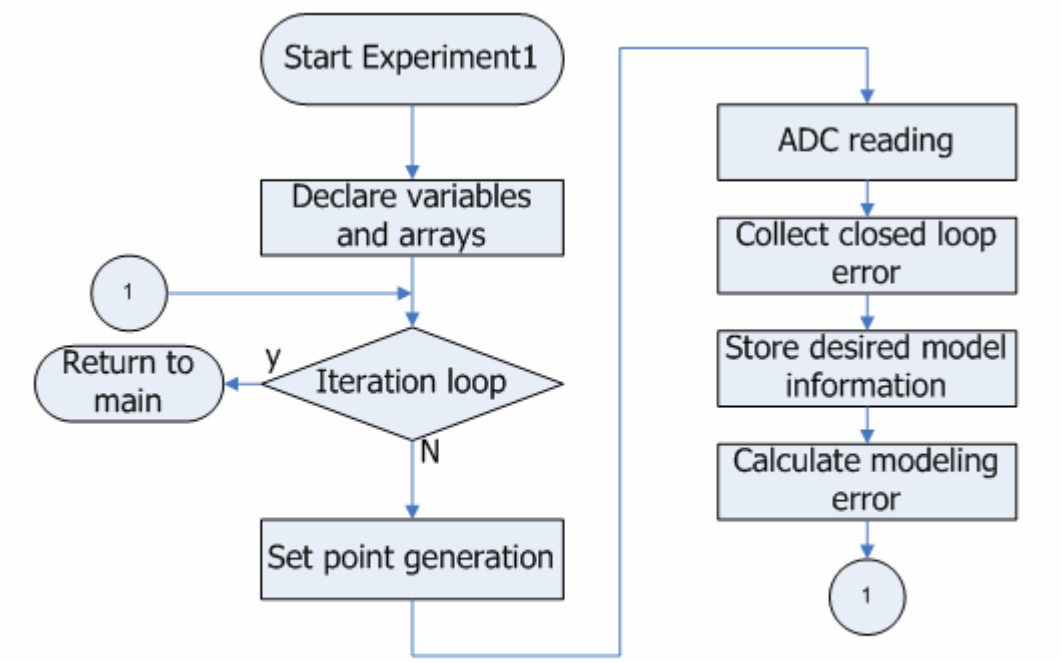

Figure 4.3 IFT Experiment1 Flowchart

The first block of the flowchart initializes variables given in table AP4.1 (Appendix Four) though modified in a quest to shrink the algorithm to the size executable into the DSP56F807C microcontroller. All arrays except for closed loop error, eta[j] and modeling error, emta[j] were converted to variables declared as floats. Variables used in function 'experiment1' function are as follows:

(1) Loop limit, N for fixing the number of iterations,

(2) Loop counter, j for the 'for loop',

(3) Controller parameters, alf0 and alf1 as given in tables AP4.1 and AP4.2,

(4) Output response, yta,

(5) Input signal, uta,

(6) ADC sample, Sample,

(7) Closed loop error,eta[j],

(8) Desired model, ymta

(9) Modeling error, emta[j].

The initialization block is followed by a 'for loop' block. The 'for loop' houses an array of tasks as given below:

(1) Square wave generator acting as a set point

(2) Sample ADC registers for feedback signal

(3) Collect closed loop error into an array, et1a for use as a set point in experiment2

(4) Calculate signal, uta as input to DC motor (fixed controller)

(5) Store desired model information

(6) Calculate modeling error

As the time limit, N of iteration was made bigger the better was the optimization. In [3], N was made one thousand (1000) but for microcontroller applications N was made five hundred (500) to save on memory that is limited in a microcontroller. Time of optimization is increased with big N. This is a recently discovered weakness concerning IFT algorithm as compared to MUC algorithm which optimizes directly.

(4) Procedure 'Exeperiment2'

After N-length iteration in function 'experiment1' the control is passed to function 'experiment2' for updating of controller parameters. Controller parameters are updated through minimization of collected closed loop error from function 'experiment1'. The minimization process is accomplished by the gradient working on the collected modeling error, emta as presented in the basic science of the IFT algorithm in Chapter Three. The flowchart for function 'experiment2' is given in figure 4.4.

Program structure is formulated same as in function 'experiment1' in that the first block in the flowchart initializes variables and is followed by a 'for loop' wrapping a number of tasks for N-length period required to achieve controller parameter updating.

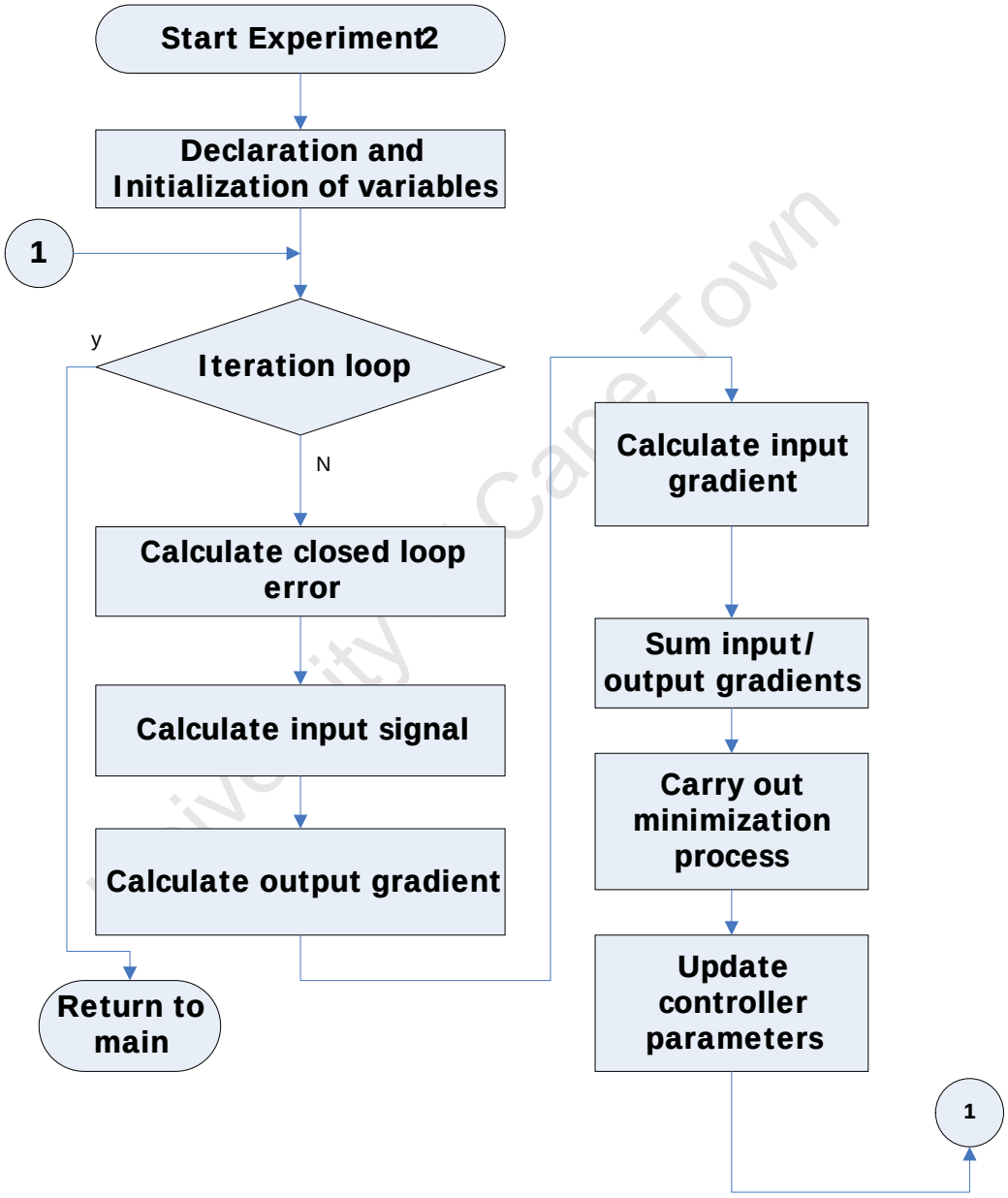

Figure 4.4 IFT Experiment2 Flowchart

In this research N was made 500 as opposed to 1000 of the visual basic IFT program in [3]. Cardinal features of function 'experiment2' are the calculation of output and input gradients with respect to controller parameters. These are then summed individually and then regrouped to form criterion minimization, J that is used for updating of controller parameters.

Function 'Experment2' loops for N-length period and then passes control to function 'experiment1' for error collection again. This goes on and on until the closed loop error is minimized and constant. Hence if there is any disturbance that may occur in the process it is identified in function 'experiment1' and then is later minimized in function 'experiment2' as already explained. After the development of IFT algorithm flowcharts as shown above, each flowchart was converted into C-language code read for implementation into the microcontroller. The code was designed to be used as compact as possible mainly for microcontroller applications and the details will be shown later.

4.2 Coding the MUC Algorithm

MUC algorithm code was a lot easier to code than the IFT algorithm code in that its sophisticated equations were reduced to simple linear algebra type. Its program structure is given in figure 4.5. MUC program structure is exactly the same as the IFT program structure except that the contents of the algorithm are different. It must be noted that the MUC code was developed using another microcontroller known as the ARM7024 (from Analog Devices) other than the DSP56F807C microcontroller initially used for development of IFT algorithm code. The reason for this shift was that MUC coding was done at the University of Zambia other than University of Cape Town and ARM7024 microcontroller was the only suitable microcontroller easily obtainable at this university. The other reason was that it was straightforward concerning the usage of the DAC as

compared to the DSP56F807C DAC that needed modification before using it. Research revealed that the DSP56F807C microcontroller has an inherent problem

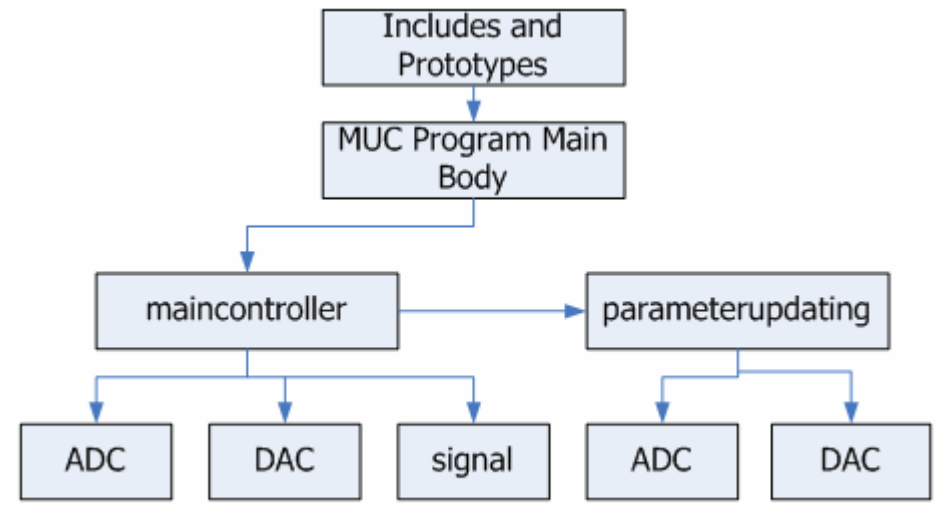

Figure 4.5 MUC Program Structure.

with its DAC, and as a result the DAC interface to the microcontroller requires extensive modification before using it [20]. A resistor or jumper must be added between DSP56F807C MISO pin and MAX5251 MISO pin to prevent the MAX5251 to drive the MISO line of the DSP56F807C. It must also be stressed that the IFT code ran well on an ARM7024 microcontroller but its usage in place of the DSP56F807C was a factor of accessibility. The difference between the two is in their processor architecture and the speed of mathematical computation. The DSP uses hardware in mathematical computations as compared to ordinary microcontroller that apply software to carryout additions. Due to the above shift, includes and prototypes for the MUC code differ from those of IFT algorithm code especially those related to the initialization of a microcontroller.

Includes and prototypes in the MUC code are listed below as follows:

(1) # include <stdio.h>

(2) #include “maincontroller.h” the header file for function ‘maincontroller ’ where input and output data are generated for testing the candidate controllers in the function ‘parameterupdating’

(3) #include “parameterupdating.h” the header file for function ‘parameterupdating’ where parameter updating takes place by gradient means when a candidate controller is falsified.

(4) #include <ADUc7024.H> the header file for ARM7024 registers such as the Analog to Digital Converter (ADC) and Digital to Analog converter (DAC) mainly used in this project.

The program main body in figure 4.5 hosts global variables (Theta, intergral, y,) and function ‘mainController’. Function ‘mainController’ calls function ‘signal’, function ‘ADC’, function ‘DAC’, and function ‘parameterUpdating’. MUC algorithm code takes no arrays as compared to IFT algorithm.

The flowchart for the function ‘main’ is shown in figure 4.6 below.

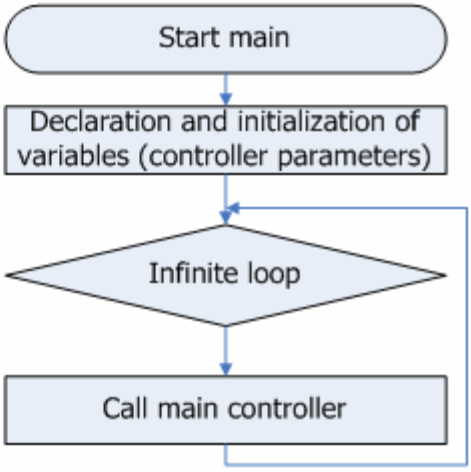

Figure 4.5 MUC Main Function Flowchart

Function ‘main’ was made to loop continuously as it was required to keep function ‘mainController’ running as a core controller.

Figure 4.7 shows flowchart for function ‘mainController’.

.

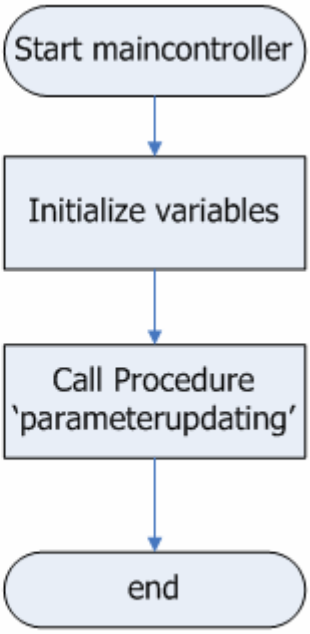

Figure4.7 MUC Main Controller Flowchart

This function calls function 'parameterUpdating' for updating controller parameters and finally, figure 4.8 shows flowchart for function 'parameterUpdating' that applies a negative gradient to the parameters whenever cost function is greater than zero.

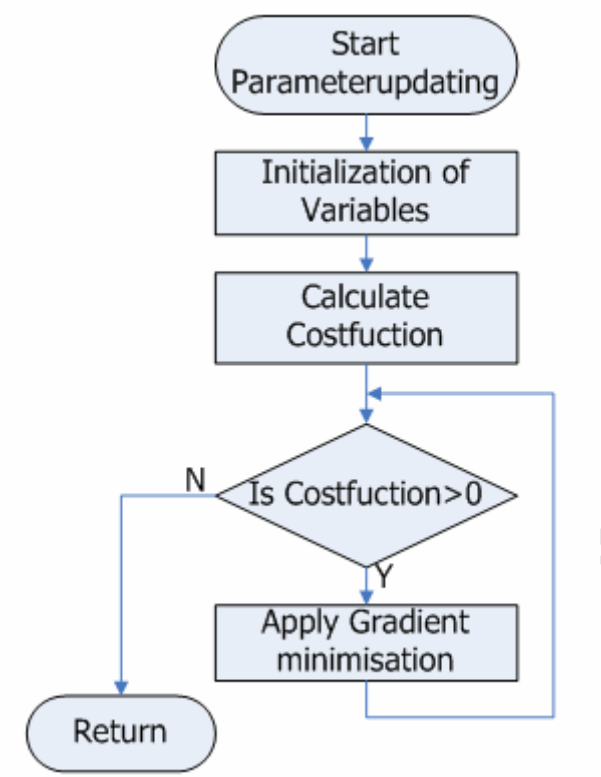

Figure 4.8 MUC Parameter Updating Flowchart

Function 'signal' is similar to the one used in the IFT algorithm while functions 'ADC', and 'DAC' are standard and depends on the type of microcontroller used.

In essence that the flowcharts had been presented based on the theory, the IFT and the MUC algorithms were coded using C-language and their respective codes are tabulated in Appendix Three.

4.3 IFT and MUC Algorithm Code in C-Language

In this section the two codes written in C- language of microcontroller hardware (DSP56F807C and ARM7024) are being formulated and discussed as it is given in the subsections below:

(1) Procedure 'Main' IFT Code

The 'N' in the two global arrays, et1a[N] and emta[N] is an iterative limit for which in this research maximum could go only up to five hundred (500) due to limitations in memory of the DSP56F807C microcontroller. The two arrays and the given variables in the main program are declared as global variables since they need to be seen in all functions of the IFT algorithm. For instance et1a[N], generated from function 'experiment1', is a set point signal in function 'experiment2' and emta[N] also generated from function 'experiment1' is needed for criterion minimization in function 'experiment2'.

The next stage was the initialization of controller parameters (proportional gain, ρ_0 and integral gain,ρ_1) for which the system is stable. Based on existing theory of boundaries for the domain of the controller parameter [3], the controller parameters used in the IFT code imbedded into microcontroller hardware were started at 0.003 and could be varied up to 1 as in [3]. Outside this domain, the output response obtained was unstable. Parameter initialization was followed by an infinite 'for loop' so that the main function scans continuously the functions imbedded in it as already explained above.

Features like program interface, and greetings were not included in this code for the sake of economizing in memory which was crucial in the sense that the license used in this research for the Motorola microcontroller was limited to eight (8kb) kilo bytes only.

(2) Procedure 'Experiment' Code for IFT

Procedure 'Experiment code' combined two functions, experiment1() and experiment2() in one file so that only one header file (experiment.h) was included in main file since the two functions were very much related in their operation in the implementation of IFT algorithm. The other reason of putting these two functions in one file was for ease of reference during testing the algorithm on microcontroller. The code for experiment header file was written as indicated in Appendix Three.

Function 'experiment1' as already explained above, constitutes a fixed controller, and collects closed loop error and modeling error required as set point to experiment2 and criterion minimization respectively.

As per the C-language format all variables were declared before using them in the actual function. Though the IFT algorithm assumes that the model of a plant is not known [4], model variables, 'am' and 'p' were initialized to suit the desired model criteria. This technique was adopted to simplify the investigations as per the objective of the research. After initialization of variables, the function enters an iteration phase. Here closed loop error and modeling error data points are collected for the N length period. The bigger the N the better the optimization but for implementation on a microcontroller freedom of varying N was restricted to a maximum of five hundred for economizing the usage of memory. While collection of data points is in progress, the fixed controller operates on initial values of controller parameters until that time when the optimum or updated parameters are obtained.

The loop begins by calling a function 'signal' that generates a set point signal, rt and this signal is a square wave. The next function to be called is the ADC

function that links the output of the process plant to the IFT algorithm. Only one input of the ADC was used since the investigations were for a single loop.

The IFT algorithm follows the ADC function for implementation of the control action. It outputs the control signal, uta to the input of the process plant through the DAC, the last function called in this loop. The sequence is then repeated for N length period before the control is passed back to the main function for calling function ‘experiment2’.

Function ‘experiment2’ has a similar format for its program structure as that for function ‘experiment1’. They only differ in application as already explained. The code for ‘experiment2’ is also given in Appendix Three.

Function ‘experiment2’ minimizes the collected closed loop error from ‘experiment1’ as a result of disparity in the desired model and the existing plant due to external disturbances. Error minimization is achieved through negative or positive gradient applied to the collected closed loop error with respect to controller parameters. For the first degree of freedom controller adopted in this research only two parameters are updated.

Rj matrix was made the identity matrix without any loss of generality. Though the value of the scalar Gama matrix has an adverse impact on the convergence properties of the IFT algorithm [3] it was also made unit. This was done as a way of shrinking the code that was necessary to achieve the objective outlined in the research. Clever coding was implemented in order to constrain parameters in the range of 0.03 to slightly less than 1. This boundary was achieved through the updating equation explained in Chapter Three and as shown in the code for ‘experiment2’.

Other pieces of code such as function 'ADC' for reading the output signal from the process, function 'DAC' for feeding the process and signal as set point to 'experiment1' are also depicted in Appendix Three:

All pieces of code dealt with above and depicted in Appendix Three and in combination implements an IFT algorithm on microcontroller Motorola DSP56F807C. This code was tested and ran without memory overflow and to prove the workability of the code the simulation results for output response from 'experiment1' and set point to 'experiment2' generated from 'experiment1' are shown in Chapter Five. The results take the same shape as those in [3, 4] that are reproduced and fixed in Chapter Five for comparisons.

(3) Procedure 'main' Code for MUC

The C- code for the MUC algorithm is given in Appendix Three. Its code is shorter than that for the IFT algorithm and is easily implemented on a microcontroller. MUC is also fast to converge as will be shown later. The concept and style of MUC coding has been dealt with in section 4.2 above. In this section only key code features are highlighted. As shown in Appendix Three, MUC code was broken down into two program files: The 'main' and the 'muc' file. Declaration of global variables and functions calling routines were fixed in this file. The reason for making the variables 'Theta, u, y, and val' global were to enable them accessible to other functions in need of such data. In the MUC main body program the 'main' function houses the infinite loop created by the use of 'while' command which in turn nest calling functions 'mainController' and 'parameterUpdating'. Control from the main file is passed to muc files by means of function calling. All the required functions in this algorithm are resident in this file. These functions are outlined below:

(1) mainController()

(2) parameterUdating()

(3) signal()

(4) ADC()

(5) DAC()

Function ‘signal()’ was created and coded for square wave generation for use as a set point while the two functions ‘mainController() and parameterUdating()’ were developed to implement the MUC algorithm itself. This code was loadable into ARM7024 microcontroller without any constraints. The reason of using ARM7024 in place of DSP56F807C has been highlighted already. The code was simulated by utilization of a fictitious plant that was inserted into the program. The plant that was used in the simulation was a first order one. Running the program and printing out output values from variables y, costFunction, Theta, and u, the graphs were plotted using excel as will be shown later.

Chapter Five.

Testing IFT and MUC Algorithm Code Implemented in Microcontroller

The code for the IFT and MUC algorithm developed in Chapter Four and implemented on the microcontroller is tested in this chapter. The test is meant to answer a research question outlined in research objectives concerning the feasibility of implementing an IFT and MUC algorithm on microcontroller hardware. This was done through:

(1) checking if the code ran on the DSP56F807C microcontroller without causing memory overflow and other errors

(2) checking signals for both IFT and MUC whether they conform to the previous results obtained by others. The IFT signals that were checked were closed loop error, et1a (collected from procedure 'experiment1') and output response, yta for procedure 'experiment2'. The MUC signals checked were output response, y, control signal, u, and the cost function, costFunction.

(3) checking the behaviour of cost function, J, in the process of parameter updating, whether minimization was taking place in modeling error.

The above signals were tested through carrying out a simulation of both the IFT and MUC algorithms by inserting a software-based plant in the IFT or the MUC code that was embedded into the microcontroller hardware.

5.1 IFT and MUC Hardware Implementation

The IFT and MUC hardware implementation is an IFT and MUC algorithm code in C language embedded into DSP56F807C or ARM7024 microcontroller. The block diagram for the IFT or MUC hardware configuration setup depicted in figure 5.1 was used to test the IFT and the MUC though for simulation purpose the DC Motor, ADC and DAC were implemented as software-based devices.

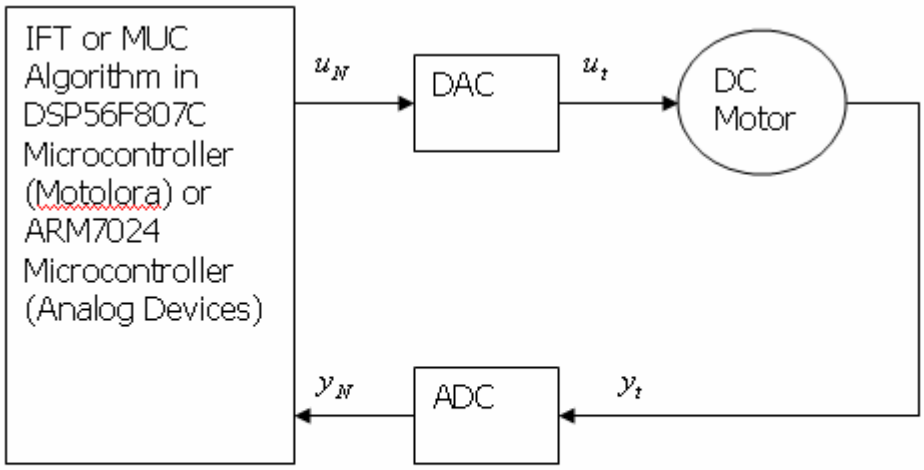

Figure 5.1 Block Diagram of IFT or MUC Hardware Implementation.

The DAC is on-board while the ADC is embedded into the DSP56F807C microcontroller. In the case of ARM7024 both the ADC and the DAC are embedded into the microcontroller. The output from the IFT or the MUC algorithm is applied to the DC motor through the DAC and the output from the DC motor is fed back to the IFT or MUC algorithm through the ADC. The DC motor is adopted as a software plant for testing of IFT algorithm embedded into the DSP56F807C microcontroller because it is widely used in industrial control and is available in the laboratory.

5.2 The IFT and MUC code into the Microcontroller Hardware

Both the IFT and MUC code ran successfully on the microcontroller hardware without causing arithmetic overflows. This was achieved by careful coding of the two algorithms and initializing certain variables in order to operate the algorithm within stable boundaries. Cardinal variables that require initialization are:

a) Controller parameters, ρ_0 and ρ_1 are normally chosen by the user bearing in mind the range between 0.03 and 1. Previous research has proved that

the given range is workable [3]. If the parameters were started at lower values less than 0.03 there was a tendency of getting undefined response from the controller. This was due to the expressions meant for calculating input and output gradient such as the one given in equation (3.15). Since the parameter is in the denominator and if it becomes small approaching zero the microcontroller displayed an undefined response (nan).

b) The matrix, R_j generally has constant elements that can have any values as long as it remains a positive matrix hence for simplicity sake and to save memory the identity square matrix with diagonal elements equals one as shown below was chosen.

$$R_j = \begin{bmatrix} 10 \\ 01 \end{bmatrix}$$

c) Iteration limit, N was started at lower values such as 20, again this was required for saving memory otherwise 1000 was recommended in [3] to ensure that enough data (closed loop error and modeling error) were collected for optimization in experiment2. As already highlighted the maximum number of iterations used in this research was five hundred (500), exactly half way of the recommended, for ARM7024 and the DSP56F807C microcontroller.

In the MUC algorithm, variables that require initialization are the controller parameters (Theta and integral), and initial cost function. These are chosen by the designer. In this research the initial cost function was set to zero and the controller parameters could be varied from 0.03 to 50 for gain and from 0.03 to 20 for integral gain without causing unstable condition in the response. The parameter range for the gain and integral only demonstrate the spectrum of

initial values for the controller and these were arrived at by experimentation. Making the initial parameters too small caused undefined condition due to dividing by zero both in the cost function and parameter updating expression. On the other hand if the initial parameters were started at any number greater than 50 for gain and 20 for integral there was a tendency of undefined response. Theoretically high gain increases speed and reduces the error of a system response but if it becomes too big the system response gets overshoots with oscillations and eventually can explode into instability.

There was the issue of introducing scaling factors to the cost function expression to avoid memory overflow in the microcontroller. The expression (costFunction = -costfini + (1/(Theta + 1)^2)*u;) was divided by one thousand (1000) to avoid making its value too big or rather cause undefined response (producing numbers such as 'nan').

5.3 Comparison with Previous Results

Since the objective of this research was to investigate the feasibility of implementing an IFT algorithm into the microcontroller hardware, the VB code from [3], also highlighted in Chapter Four was translated into C-language of the microcontroller so that the results obtained in this research would be compared with those of [3] in order to establish the possibility of implementing the IFT algorithm into microcontroller hardware.

The model used in this research for this particular test was the same one utilized in [3] (as is indicated in equation (5.1)).

$$G = \frac{1.01}{1+2s}\,[v]/\,[v] \qquad (5.1)$$

This model was used in [3] at the sampling time of 50ms and the same was used as the sampling time in the microcontroller hardware to generate results required for comparison with those of [3].

Equation (5.1) is converted to pulse transfer function

$$\text{gh (z)} = \frac{1.01(1-e^{-0.05/2})}{z-e^{-0.05/2}}\,[\text{v}]/[\text{v}] = \frac{0.0249}{z-0.97531}\,[\text{v}]/[\text{v}]$$

$$\text{gh(z)} = \frac{0.0249z^{-1}}{1-0.97531\text{z}^{-1}}\,[\text{v}]/[\text{v}] \qquad (5.2)$$

and the digital equation is

$$y_n = 0.97531y_{n-1} + 0.02490u_{n-1}\,[\text{v}] \qquad (5.3)$$

The closed loop model is obtained using the closed loop expression below and letting

k(s) = 1

$$\text{y(s)} = \frac{g(s)*k(s)}{1.+g(s)*k(s)}*r(s)$$

This turns out to be

$$\text{y(s)} = \frac{0.502488}{1+0.995025s}*r(s) \qquad (5.4)$$

Changing to pulse transfer function as above yields

$$\text{gh(z)} = \frac{0.502488*(1-e^{-0.1/0.995025})}{z-e^{-0.1/0.995025}}\,[\text{v}]/[\text{v}] = \frac{0.048045}{z-0.902488}\,[\text{v}]/[\text{v}]$$

$$\text{gh(z)} = \frac{0.048045z^{-1}}{1-0.902488\text{z}^{-1}}\,[\text{v}]/[\text{v}]$$

and to digital control equation

$$y_n = 0.902488y_{n-1} + 0.048045u_{n-1}\,[\text{v}] \qquad (5.5)$$

The results obtained are depicted in figure 5.2 and those of [3] in figure 5.3 to allow direct comparisons. Figure 5.2 shows two graphs and the figure just below it is the output response, yta from experiment1. The purple graph in figure 5.2 is

the desired model and the graph in blue is the output response from experiment1.

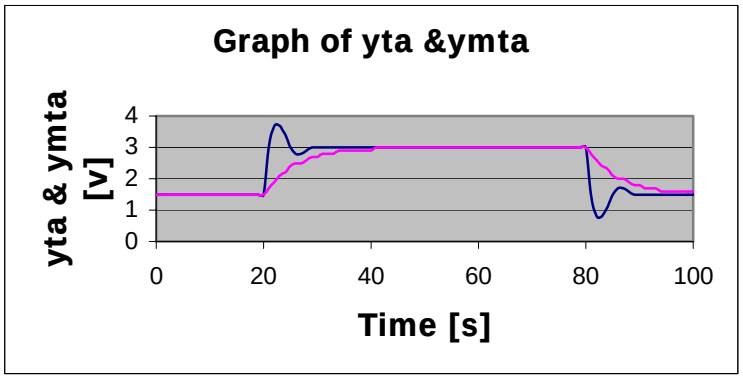

Figure 5.2 IFT closed loop response, yta & ymta

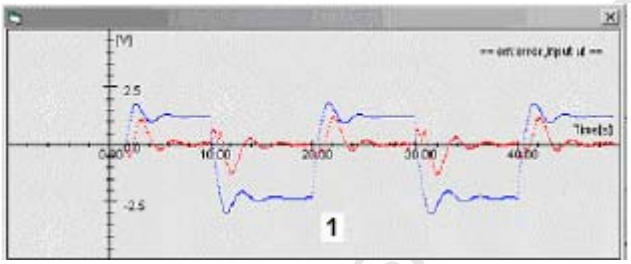

Figure 5.3: The results from experiment1 for the IFT algorithm [3]

The overshoot is 0.5v which is the same as the overshoot of the graph from [3]. The graphs are almost the same in shape. Their settling times are almost the same (the graph from [3] has 7s and the graph for this research has approximately 9s).

Figure 5.4 and the figure 5.5 (from [3]) indicate output response, yta from experiment2. The gap between responses in figure 5.4 illustrates the period of experiment2. The graphs compare almost the same in shape, overshoots and settling time.

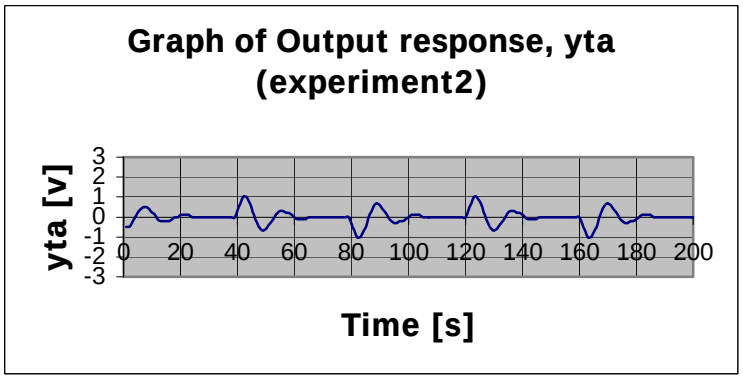

Figure 5.4 IFT output response, Yta from experiment2

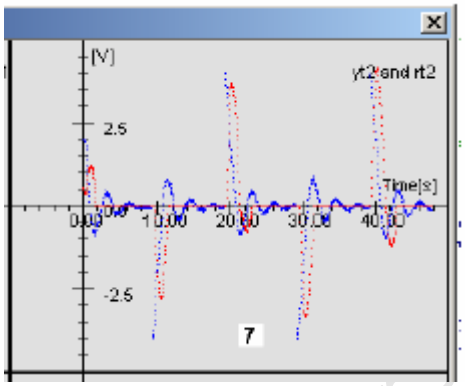

Figure 5.5:The results from experiment 2 for the IFT algorithm [3]

Similarly, the comparison was conducted for the MUC algorithm utilizing the same model from [23] as indicated in equation (5.6), to demonstrate the possibility of implementing MUC algorithm into microcontroller as was done for IFT previously.

$$g(s) = \frac{1}{s*(s+2)} \text{ [v/v]} \qquad (5.6)$$

And the following were used same as in [23]:

Performance specification, $T_{spec} = (\tilde{r} - y)^2 / 2$;

$k(\theta) = \theta > 0$; adapt ion rate, γ=5

finite time window with maximum length, T, is not less than 10s and initial cost function is $\rho(t) = e^{-1.5t}$, and the sampling time is $\Delta t = 100ms$.

Equation 5.6 is then changed to pulse transfer function

$$gh(s) = \frac{0.5 * z * (1 - e^{-0.2})}{z^2 - z * e^{-0.2} - z + e^{-0.2}} * [v/v] = \frac{0.090635z}{z^2 - 0.181269z + 0.818731} [v/v] \quad (5.7)$$

$$gh(s) = \frac{0.090635 * z^{-1}}{1 - 0.181269 * z^{-1} + 0.818731z^{-2}} [v/v]$$

and to a digital control equation

$$y_n = 0.181269y_{n-1} - 0.818731y_{n-2} + 0.090635u_{n-1}[v] \quad (5.8)$$

The results of the test are demonstrated in figure 5.6.

Figure 5.6 depicts two graphs of MUC control input, u placed side by side. The graph on the left is from [23] and on the right is for this research. The two graphs compare well in shape, frequency of oscillation and settling time. Hence it is deduced that MUC algorithm can work as expected when implemented into microcontroller

r.

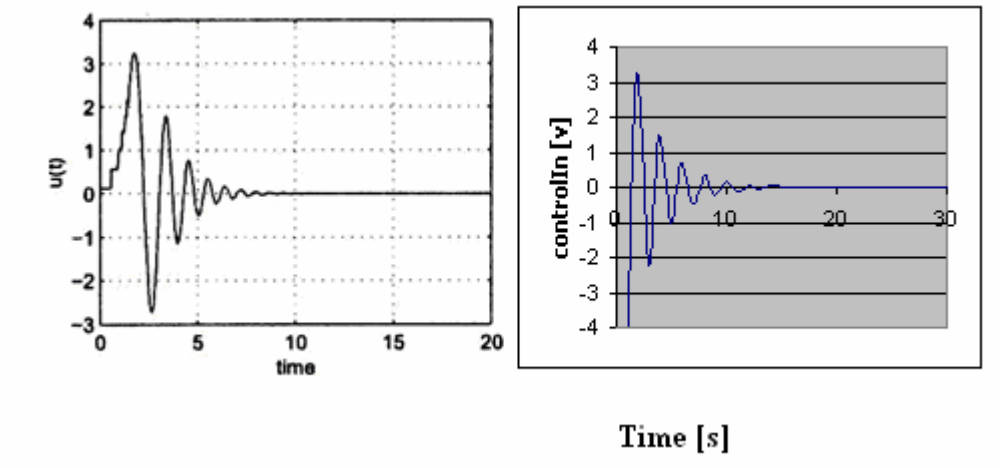

Figure 5.6 The MUC control input, u (Left graph from [23])

5.4 Modeling Error Convergence

The other test used to evaluate the IFT and the MUC code imbedded into the microcontroller hardware (to show that it is working as expected) was by checking the behaviour of the modeling error for the IFT code and cost function for the MUC code. The graphs are illustrated in figures 5.7 for IFT and 5.8 for MUC. For the IFT algorithm modeling error is expressed by comparing the two responses, output response, yta and the desired model, ymta.

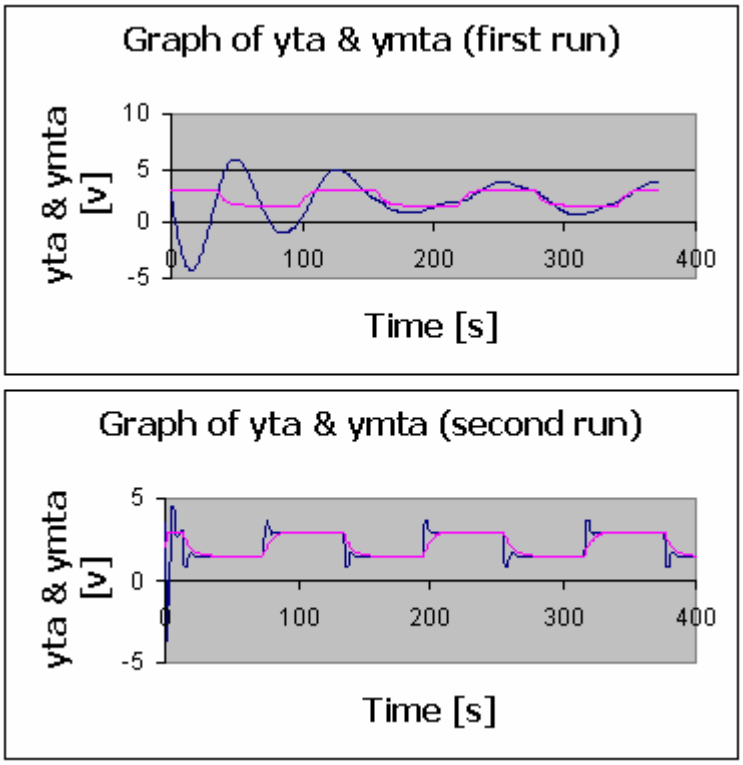

Figure 5.7 IFT Output response and desired model, yta & ymta

The graph of yta & ymta (first run) in figure 5.7 illustrates the responses of the yta in blue and the ymta in purple. The error between the two graphs is large in the beginning of the response and reduces towards the end of the graph suggesting that optimization has not yet started in the beginning and as it starts

error reduces. This is confirmed in graph of yta & ymta (second run) that indicates further reduction in error but never comes to zero. The yta begins with a slow response as illustrated in figure 5.7 (first run graph) with reduction in modeling error and then it becomes fast in the second run but with an overshoot of 0.5v that settles within 10s to the same level voltage of the desired model suggesting that at this time modeling error reduces and gives a tendency to move towards convergence which is required in IFT. This reduction in modeling error demonstrates the working of the IFT algorithm, imbedded in a microcontroller, as expected.

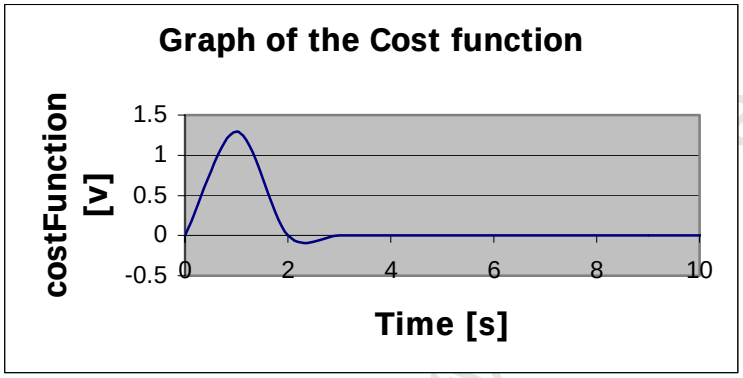

Figure 5.8 MUC Cost function, costFunction

Having verified the working of IFT algorithm on to the microcontroller through the use of modeling error analysis the working of MUC algorithm is also tested utilizing the same model from section 5.3. With necessary initialization of other variables (costFunction, Theta, integral, costFuini etc.) the graph of the cost function converged to 0 which is true in the sense that falsified controller is not discarded but adjusted in the steepest-descent direction so that performance specification, $\tilde{J}(i,t) \leq 0$ (i is controller parameter vector) is always satisfied. That is to say adjustment will only occur if and only if performance specification

expression $\tilde{J}(i,t) \leq 0$ is not true and as long as it is true cost function converges to zero.

5.5 Determining the Model of the DC Motor

Though the IFT algorithm assumes that one does not know the model of the plant being controlled, for this section of thesis however, it is assumed that the plant is known in order to ascertain whether the IFT or the MUC algorithm runs on the DSP56F807C microcontroller.

The model for the DC motor used for testing in this research was derived in order to approximate the settling time required to set the sampling time of the ADC in the microcontroller and the timing of the IFT and MUC code. On the other hand the derived model would set a base in which the results of the actual physical device would be compared with. This was done from a series of step responses that were carried out on to the DC motor (Feedback DCM 150F) and the graphs are in indicated Appendix Five, figure AP5.1 for three tests. The general first order expression shown below (5.9) was completed by substituting the averaged values for the gain, A, and time constant, T, that are tabulated in table2.

$$g(s) = \frac{A}{1+sT} \tag{5.9}$$

The model obtained out of the data given in table2 (for DC Motor with light disc) is given in equation (5.10).

$$g(s) = \frac{17.33}{1+1.16s}\ [V]/[V] \tag{5.10}$$

Another approximation of the model that was derived is of the DC Motor with heavy disc. Only two-step responses were collected and are tabulated in table3 below. The model yielded from the data in table3 is shown in equation (5.11).

$$g(s) = \frac{17.75}{1+6.7s}\ [V]/[V] \tag{5.11}$$

Table2 Step Response of the DC Motor for the light disc

Parameter	Step test1	Step test2	Step test3	Step test4	Step test5	Average	Standard deviation
T(test1)	1.28s	1.0s	1.36s	0.72s	1.44s	1.16s	0.297s
A(test1)	17.43	17.3	17.33	17.23	17.35	17.33	0.073
T(test2)	1.32s	1.56s	1.32s	1.88s	1.28s	1.47s	0.254s
A(test2)	17.28	17.75	17.25	22.35	19.9	18.91	2.212
T(test3)	1.28s	0.28s	1.28s	1.68s	1.28s	1.16s	0.522s
A(test3)	17.55	17.23	17.33	17.38	16.98	17.29	0.210

Table3 Step Response of the DC Motor for the heavy disc

Parameter	Step test1	Step test2	Average	Standard deviation
T(test)	5.8s	7.6s	6.7s	1.273s
A(test)	17.7	17.8	17.75	0.071

Equations (5.12) and (5.13) were then converted to pulse transfer functions

$$gh(z) = \frac{1.431382z^{-1}}{1-0.917404z^{-1}} \tag{5.12}$$

$$gh(z) = \frac{0.262958z^{-1}}{1-0.985185z^{-1}} \tag{5.13}$$

and to a difference equations used in digital control

$$y_n = 0.917404\,y_{n-1} + 1.431382\,u_{n-1} \tag{5.14}$$

$$y_n = 0.985185\,y_{n-1} + 0.262958\,u_{n-1} \tag{5.15}$$

These models were derived using sampling time of 100ms in order to achieve the timing period of 50s since each procedure 'experiment' cycled through five hundred times. The 32s timing was recommended for the control of the DC Motor (heavy disc) since it required approximately 8s to reach the 63% of the damped response and this time constant was multiplied by four for the output to settle within 2% of the final value or steady state value. Then the 18s left acted as a delay between experiment1 and experiment2 to ensure the action to the motor was fully exhausted.

The sampling time that achieves 32s fixed by the DC Motor with 500 samples, as the maximum in the DSP56F807C and ARM7024 microcontroller, is 64ms but it was necessary to provide additional time (another 32s) to utilize the motor action from the controller completely before another experiment is commenced. But creating long delays within program loops was not preferred since it caused discontinuities in the program flow. This was resolved by having longer sampling time such as 100s as given above to cater for the delay loops as well. Another choice sampling utilized was 86ms resulting in 43s settling time more than the actual 32s. This can make sure the settling time is covered in case of changes in the physical system. The 86ms was used to time the microcontroller as will be shown in the next section.

5.6 Timing the Microcontroller

The DC Motor used in this research requires 32s of settling time to reach 2% within of the steady state value. There was need therefore to time the IFT and the MUC algorithms running on the microcontroller in hard real time situation to suit with the DC Motor settling time selected for use as the plant in this research.

The timing exercise was accomplished through inserting the software generated ramp signals (illustrated in equation (5.16)) in both experiment1 and experiment2. These ramp signals ran for 40s in experiment1 and 43s in

experiment2. The two experiments were separated by a delay of 38s making a total IFT loop time 121s. The graph that shows the timing is given in figure 5.9. The timing that is given here was applied to both IFT and MUC algorithm.

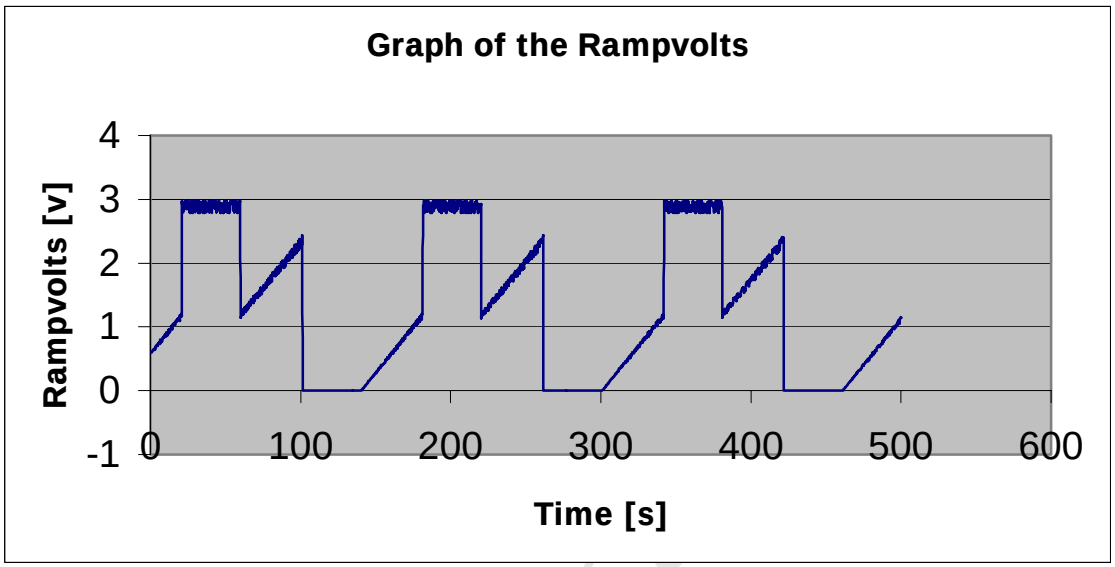

Figure 5.9 Real time of IFT Algorithm on Microcontroller

Rampvolts = Rampvolts + 1/500 (5.16)

The added fraction (1/500) creates a ramp signal after 500 cycles as shown in figure 5.9. The starting point of the generated ramp signal was normally the initial value of the valuable 'Rampvolts'. In this case, and before the beginning of experiment1, the initial value for 'Rampvolts' was made 0v and at the end of experiment1 'Rampvolts' was forced to 3v. This resulted into the Ramp signal shooting up to 2.8v and this voltage level maintained for approximately 38s with superimposed noise which may have come from the power supply for the microcontroller board. After the 38s delay time, 'Rampvolts' was initialized to 1v and began ramping up for the period of experiment2 which took about 43s. with completion of the IFT cycle, 'Rampvolts' was again initialized to 0v and delayed for 40s as in the first instance. Again, the same loop was repeated for 500 times.

Chapter Six

Application of IFT and MUC Algorithms to a DC Motor

Having confirmed that the IFT and MUC algorithms run on microcontroller hardware and the results obtained compared well with the results of others in [3] and [23] these are now applied to a DC motor model obtained in section 5.4. The results of the simulation can then be compared with the results of the actual application of the DC Motor to further evaluate the feasibility of implementing the IFT and MUC algorithm into the microcontroller hardware.

6.1 Simulation of IFT control of the DC Motor

The IFT control of DC Motor models obtained in section 5.4, for both heavy and light disc is simulated here. The graphs of the output responses of the said models with their desired models are depicted below and the graphs of their respective input signals are indicated in Appendix Six.

The heavy disc model was simulated first and the details of various models are being reproduced here for ease of reference.

The process model is g(s) = $\frac{17.75}{1+6.7s}$ [v/v]

The desired response model is m(s) = $\frac{1}{1+4s}$ [v/v]

The controller was initialized to k(s) = $\frac{0.02_{+}0.04s}{s}$ [v/v]

and subsequently converged to k(s) = $\frac{3.0_{+}3.0s}{s}$ [v/v]

The two responses of desired model, ymta in purple and the output response, yta in blue are indicated in figure 6.1. The responses were obtained from the two runs (i.e. a run is a complete cycle of the IFT algorithm) of the IFT algorithm.

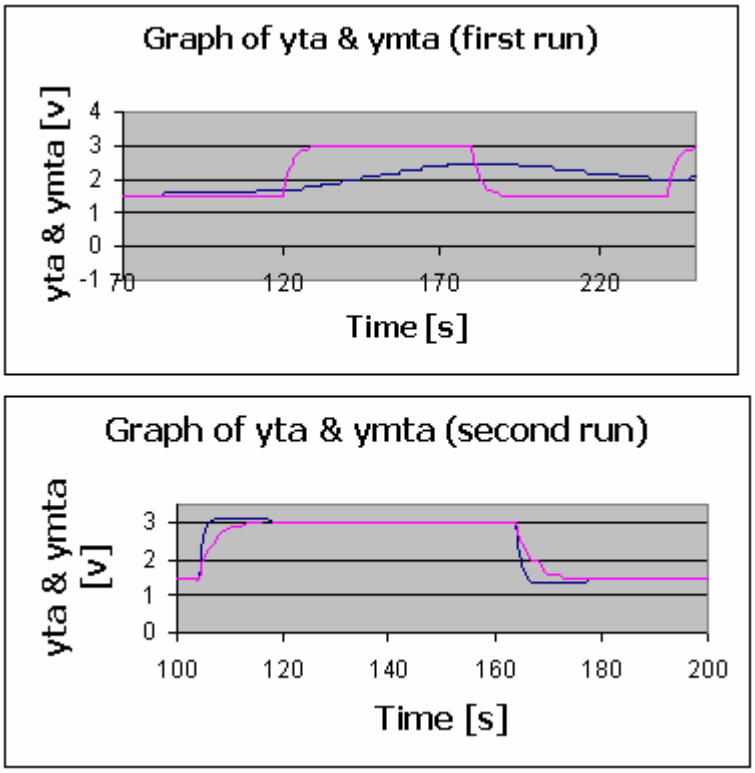

Figure 6.1 Output response, yta & ymta of the DC Motor (heavy disc) simulation.

In the first run Ymta is fast and damped with a settling time of 10s while yta is slow and damped having a settling time of 54s. the closed loop is approximately 1v. This is true since the heavy disc model has its pole closer to the unit circle in the z-plane compared to the pole of the desired model. The most important point here is the action of the plant adapting to that of desired model a key concept of the IFT algorithm. This phenomenon begins to manifest itself in the second run of the IFT algorithm because of the action of the IFT.

In the second run of the IFT algorithm, yta becomes faster than ymta but settles 4s latter than the desired model (at approximately 14s) an improvement from the response of yta in the first run which takes 54s to settle. This is a

demonstration that controller parameters are being optimized in relationship with gradient minimization of modeling error, emta collected from experiment1.

Notice that controller parameter range specified in [3] was overshot by 2v without the output response getting into an unstable condition. Hence it can be deduced that the controller parameter range to be utilized by the IFT depends on the model used but not tied to the range (0.03 to 1) found in [3].

The other problem identified was the changing of already optimized parameters (from function 'experiment2') by the program in function 'experiment1'. In order to resolve the said problem there is need to modify IFT algorithm so that it should only optimize when there is a presence of modeling error as it is done in the MUC algorithm. This can help to maintain the already optimized parameters constant until there is un deterministic disturbance in the system through changes in the model and noise sources.

But on the other hand it is observed that the heavy disc model did not converge to the desired model. The reason could be the problem of non-minimum phase pole-zero cancellation that occurs especially when the desired model is fixed and not let to adjust as the controller parameters are being adapted. This phenomenon is easily explained using a PI controller given below:

$$k(s) = \frac{\rho_{0+}\rho_1 s}{s} \text{ [v/v]} \qquad (6.1)$$

$$\text{Hence } k(s)*g(s) = \frac{\rho_{0+}\rho_1 s}{s} * \frac{1}{1+Ts} \text{ [v/v]} \qquad (6.2)$$

Equation (6.2) can result in pole-zero cancellation if $\frac{\rho_1}{\rho_0} = T$ since

$$k(s)*g(s) = \frac{\rho_0(1+\frac{\rho_1}{\rho_0}s)}{s} * \frac{1}{1+Ts} = \frac{\rho_0}{s} \text{ [v/v]} \quad (6.3)$$

pushing the pole to the origin therefore making it an integrator and this was the case since the parameters were equal after convergence. This explains the outcome of the response being very fast and not materializes as expected. It entails then that even with many IFT runs the plant model would not have converged to the desired model.

The light disc model was also simulated to compare with the heavy disc model in which case later one model would be selected for comparison with the actual application of the DC Motor. It must be stressed that out of the three light disc models derived (in Chapter Five), one was chosen in respect to its balanced standard deviation in both time constant and gain. For the same reason the light disc model was selected as the plant for the actual physical application.

The details of various models are being reproduced here for ease of reference.

The process model is $g(s) = \frac{17.33}{1+1.16s}$ [v/v]

The desired response model is $m(s) = \frac{1}{1+4s}$ [v/v]

The controller was initialized to $k(s) = \frac{0.02_{+}0.04s}{s}$ [v/v]

and subsequently converged to $k(s) = \frac{0.6_{+}0.6s}{s}$ [v/v]

Figure 6.2 illustrates two responses of desired model, ymta in purple and the output response, yta is in blue from two runs of the IFT algorithm on to the microcontroller for light disc model.

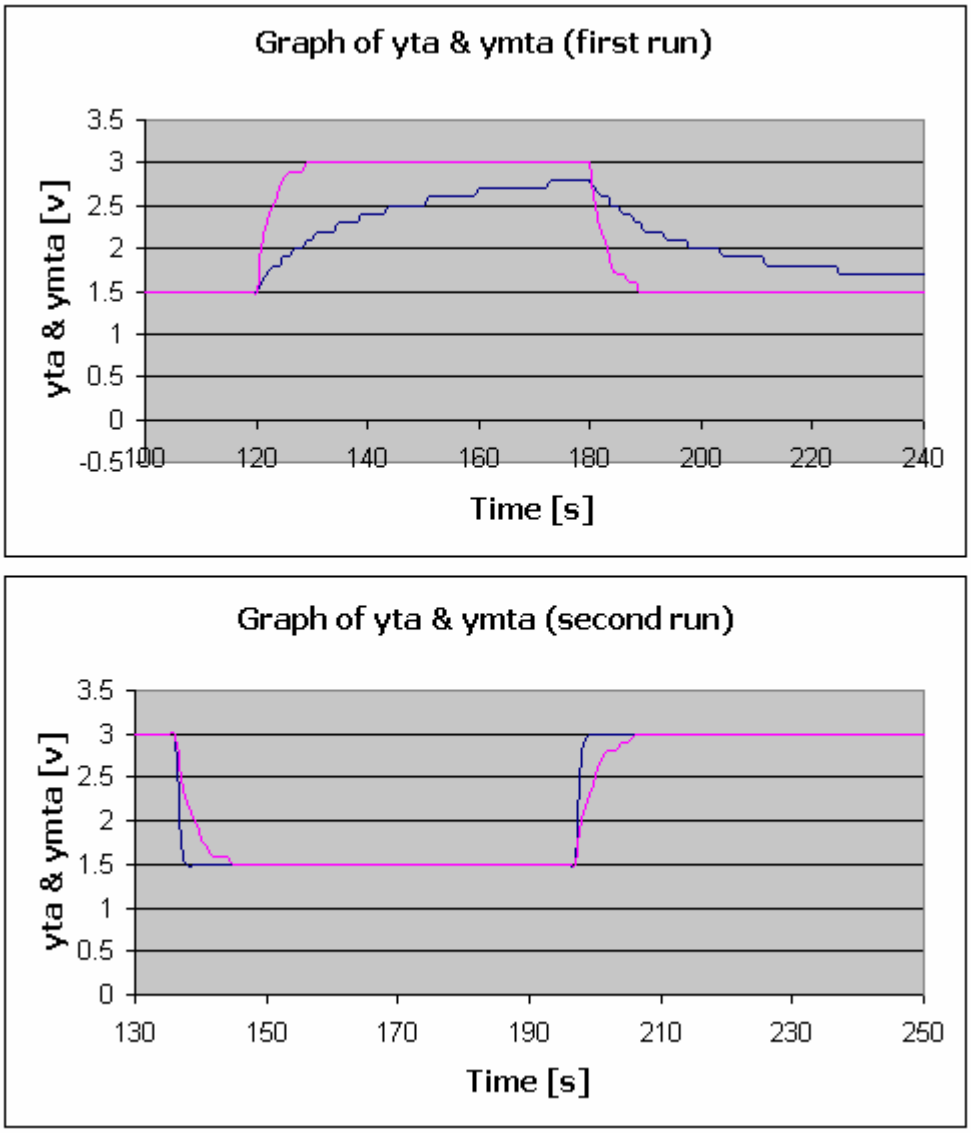

Figure 6.2 Output response, yta of the DC Motor (light disc) simulation

In the first run a similar scenario is observed, yta is slower than ymta settling within 54s. It does not converge to the final value of ymta but its convergence value (2.8v) is much better than the heavy disc situation. There is no sign of yta adapting to ymta model demonstrated in that the difference between the two models after the first run is maintained at 2.8v final value as compared to 3.0v final value of ymta.

In the second run of the IFT algorithm, yta becomes faster than ymta but settles at the same time, at approximately 10s, an improvement from the response of yta in the first run which takes 54s to settle. This is a demonstration that controller parameters are being optimized in relationship with gradient minimization of modeling error, emta collected from experiment1 though the g(s) did not converge to the desired model. The reason for this problem is already dealt with above for the case of heavy disc model. This problem can be attributed to using a fixed desired model as already highlighted for the case of the heavy disc model. It is becoming clear now as in [17] that the desired model should shift as the controller parameters are being adjusted otherwise IFT will cause premature convergence.

Another observation made was that of the range of the controller parameters. There was a tendency of sweeping in the whole controller parameter region (from 0.02 to 1.0) by adapting equations to search the parameters that converge to the light disc model suggesting that for unconstrained region the parameters can get bigger and bigger as long as the modeling error does exist, hence causing overshoots or unstable condition in the output response. This problem is also as the result of fixed desired model.

6.2 Simulation of MUC control of the DC Motor

Similarly, MUC control of the DC Motor models obtained in section 5.4 for both heavy disc and light disc are simulated here. The graphs of the output responses of the said models are depicted below. As discussed in Chapter Three, MUC has no desired model in its algorithm. Its convergence is triggered by performance specification, $\tilde{J}(\theta,\tau) \leq 0 \forall \tau \in [0,t)$ when its condition is satisfied. In view of the above the results that are presented indicates only the output response, y.

The initial conditions of the simulation for MUC heavy disc model are reproduced here for ease of reference.

The process model is g(s) = $\frac{17.75}{1+6.7s}$ [v/v]

The controller was initialized to k(s) = $\frac{10.0_{+}10.0s}{s}$ [v/v]

and subsequently converged to k(s) = $\frac{4.2_{+}0.3s}{s}$ [v/v]

Figure 6.3 demonstrates the output response, y of the MUC code imbedded into microcontroller for heavy disc model.

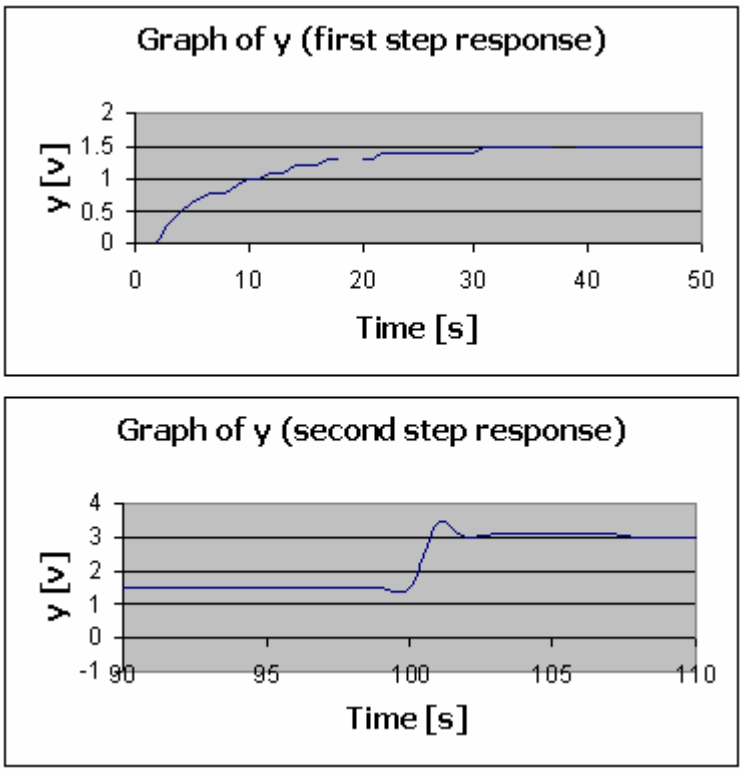

Figure 6.3 MUC Output response, y of the DC Motor (heavy disc) simulation

As seen in the output response, y of the heavy disc model, it takes y 50s to settle to its final value and has an overshoot of 0.5v. This is opposite of the occurrence in IFT heavy disc model. In the IFT control the model settled within 13s a big difference to settling time of 50s with MUC control. The deduction from this is that MUC could have started with a bad controller. Normally, MUC begins

with a set of controllers, which are tested for falsification one at a time and depending on the outcome negative gradient is applied on them. The controller used in this research was only one and could not have been the right one for the heavy disc model because the cost function converged to zero before error due to overshoot-damped oscillations were removed. This error is removed when performance specification $\tilde{J}(\theta,\tau) \leq 0 \forall \tau \in [0,t)$ is less than 0. Hence just as it is necessary to adjust the desired model in IFT it is also necessary to start with a sizeable number of controllers.

The initial conditions of the simulation for MUC light disc model are reproduced here for ease of reference.

The process model is g(s) = $\frac{17.33}{1+1.16s}$ [v/v]

The controller was initialized to k(s) = $\frac{10.0_{+}10.0s}{s}$ [v/v]

and subsequently converged to k(s) = $\frac{0.3_{+}0.3s}{s}$ [v/v]

Figure 6.4 demonstrates the output response, y of the MUC code imbedded into microcontroller for light disc model simulation.

The output response, y takes 12s to settle faster than that of the heavy disc model. It has an overshoot of 0.2v.

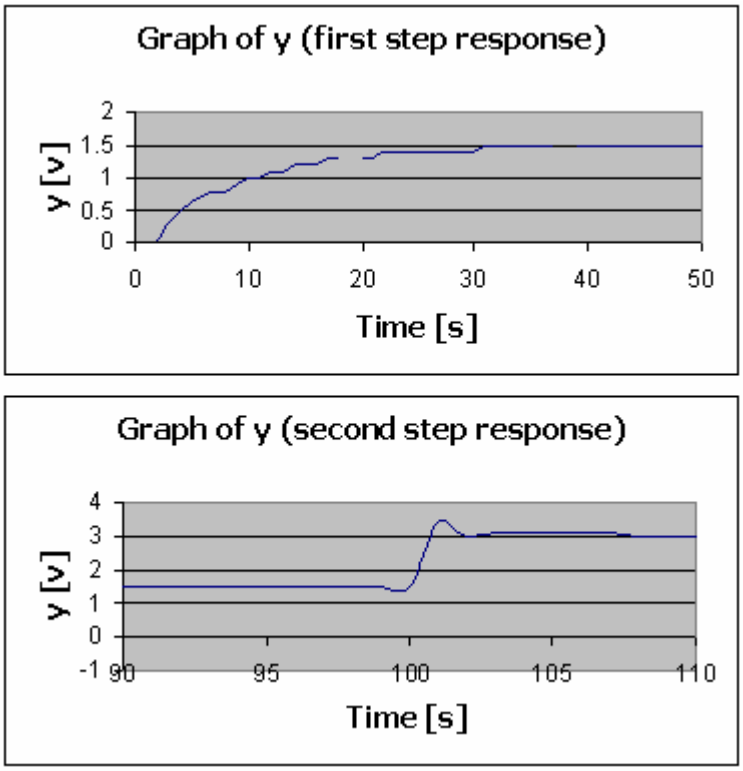

Figure 6.4 MUC Output response, y of the DC Motor (light disc) simulation

6.3 IFT and MUC Algorithm Control of the DC Motor

In this section the IFT and MUC are applied to the DC Motor and only the light disc type is considered.

The DC motor used in this research is the feedback type with tacho generator attached to it. The tacho generator was used to provide feedback to the IFT or MUC algorithm as shown in figure 6.5. The objective of the test is to prove whether the results of the physical test and that of simulation obtained in Chapter Five agree so that if there is some similarity in the outcome of the two compared results a conclusion would be drawn depending on the outcome of the comparison. The procedure of the experiment is given below.

6.1 Procedure of the Test

The set up for the test is shown in figure 6.5 and figure 6.6 magnifies the microcontroller board that was used in the implementation of the IFT and MUC algorithm.

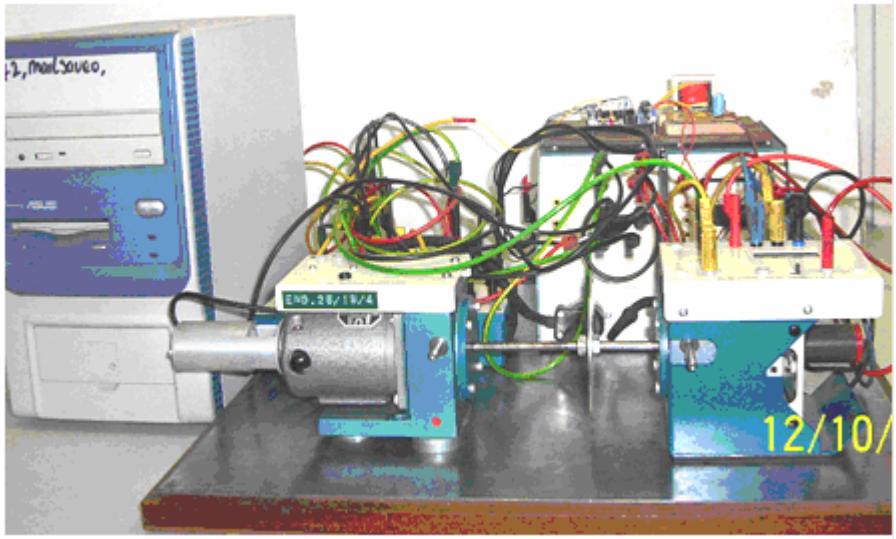

Figure 6.5 Setup for Testing the IFT and MUC Algorithm applied to a DC Motor.

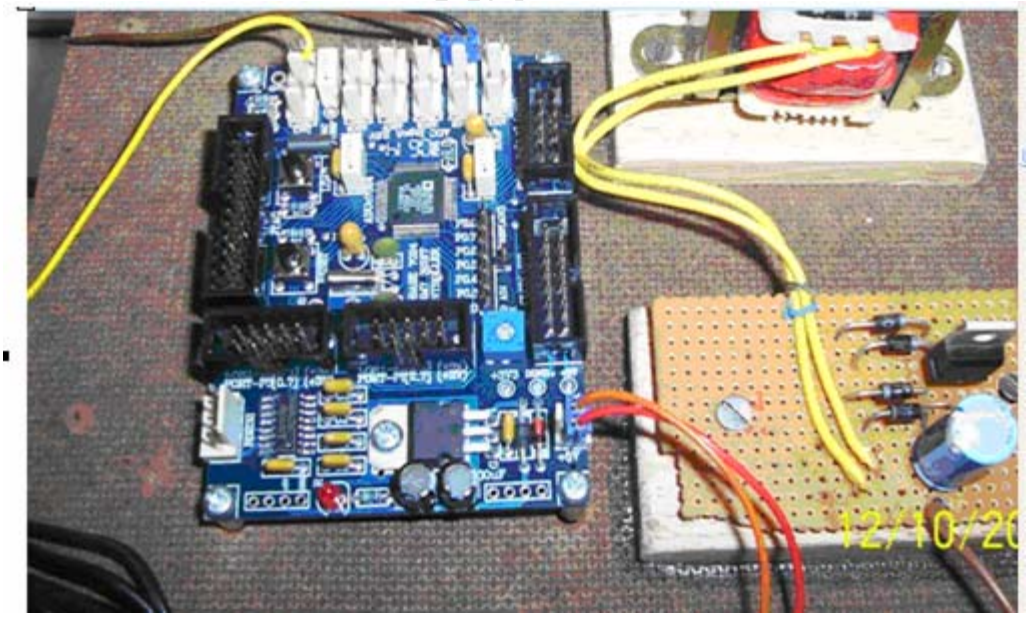

Figure 6.6 ARM7024 Microcontroller Board and Power Supply

Other instruments used in this setup were as follows:

(1) DC motor (feedback MT150F)

(2) Tacho Generator (feedback GT 150 X)

(3) Power supply unit (PS 150E)

(4) Servo Amplifier Unit (SA 150D)

(5) Attenuator Unit (AU 150B)

(6) Two operational amplifiers units (PA 150A)

(7) Microcontroller Board (ARM7024)

(8) Computer (PC)

The results of the output of the DC motor, yta for an IFT algorithm are depicted in figure 6.7.

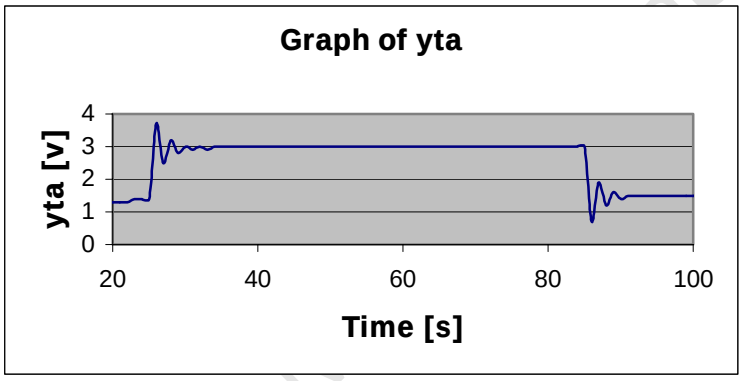

Figure 6.7 IFT DC Motor (light disc) output response, yta

Motor output response, yta from the IFT algorithm running on a microcontroller is indicated in figure 6.7. The graph illustrates a maximum voltage of 3.0v with damped oscillations and a minimum voltage of 1.5v. The DC Motor operated on upper limit voltage for the period of 60s and did the same on the lower limit voltage. The settling time is 8s almost same as the settling time of the light disc model simulation. The motor ran on two limit voltages (1.5v and 3.0v) continuously making it slow at a given period of time and fast for another period.

This behaviour renders IFT not to be the best controller for constant speed control of the DC Motor but for tuning controllers to the specification of the plant in question.

In case of MUC algorithm applied to the same DC Motor (light disc type) used in the IFT algorithm setup, the graph of DC Motor output, y is indicated in figure 6.8. This graph illustrates output response, y from two step responses. The settling time is 8s almost same as that of the simulation graph. The DC Motor ran at 3.0v and 1.5 constant consecutively.

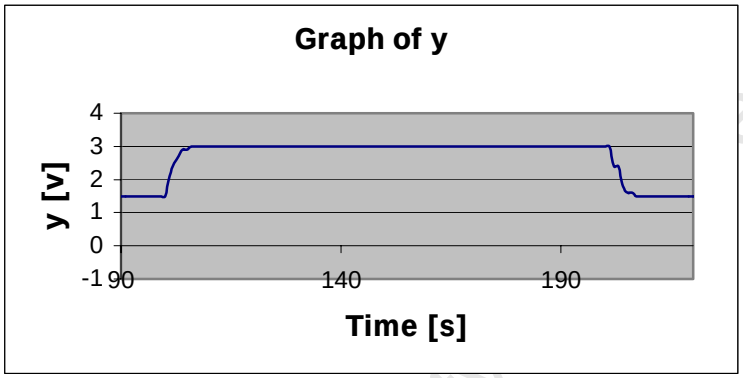

Figure 6.8 MUC DC Motor (light disc model) output response, y

Table 4 Comparisons of Simulation and Actual physical implementation of the DC Motor results

Method	Simulation	Actual
IFT (settling time)	10s	8s
MUC (settling time)	6s	8s

The simulation results and those of actual physical implementation for the light disc model as specified earlier agree in settling time as shown in the table above.

Hence it is feasible to implement IFT and MUC on microcontroller. Though IFT algorithm still needs attention, especially in preventing non-minimum phase, pole-zero cancellation.

MUC can definitely be applied for simple applications such as temperature control, speed control of motors etc. with little attention to include a set of many controllers.

Chapter Seven

Discussion and Conclusion

7.1 Discussion

IFT and MUC algorithms were implemented on a microcontroller by coding them into the C-language of the DSP56F807C and ARM7024 microcontroller. Memory spaces taken by the IFT and the MUC algorithms onto ARM7024 microcontroller were 5563 and 5595 bytes respectively representing about 4.6% of the total program memory of either of the two microcontrollers' program memory. Hence the critical issue in imbedding the two algorithms on a microcontroller is not of the program memory space but that of RAM space, especially the IFT that needs to store one thousand data points in two cycles for first degree of freedom algorithm. For example IFT made use of two arrays which were declared as global floats and in addition to other three global variables meaning that a total of 8016 bytes of memory space was required on the stack area. That is the reason only 500 points could be collected in the DSP56F807C implemented IFT controller because of limitation in the RAM space. In the case of the MUC algorithm there is only one global array that needs to collect closed loop error data for optimization. But MUC optimizes controller parameters directly. Other global variables used were twelve in number occupying a stack area of 34 bytes meaning that almost all the microcontrollers listed in Table1 (Chapter Two) can run MUC algorithm without having memory (RAM) problems.

The two algorithm programs ran on an ARM7024 microcontroller. This conclusion is drawn from the results obtained for IFT and MUC algorithms implemented on a microcontroller and showing some similarity with those obtained by others in [3 and 23].

In case of processing speeds of IFT or MUC program, these depended on the physical models of the plants used as given in Chapter Five. The model for the heavy disc has approximately a 8s time constant and the model for the light disc has got approximately 2s. These imposed a particular speed on the IFT or MUC. For example for the heavy disc model a period of 96s is required so that the two experiments takes 64s and in between them another 32s to separate the two experiments.

In general, IFT requires fast speed of processing because it gathers data for N-length time and then processes the same data for N-length again. A summary of IFT and MUC memory usage and other properties are provided in table 5.

Table 5 Summary of memory usage and other properties for IFT and MUC

Property	IFT	MUC
RAM size	8016 bytes	34 bytes
Code size	5563 bytes	5595 bytes
Execution time	100 ms	100 ms
Speed of convergence	60s (light disc model)	10s (light disc model)
Absolute errors	15.4%	14.3%
Oscillation in controller parameters	Yes, before convergence is achieved	Yes, before convergence is achieved

The heavy disc model simulation results for IFT in Chapter Six demonstrated clearly the action of adaptability of output response, yta to almost that of the desired model. As already highlighted in Chapter Six it took the whole of first run for the output response, yta without converging to a stable and fast response.

This only took place in the second run of the IFT algorithm. Hence a demonstration of the working of the IFT in the microcontroller hardware as expected.

In case of the speed of convergence, IFT takes more time (60s) to converge than MUC for light disc model. This observation concurs with the conclusion made in [21] that MUC tend to adapt more quickly than the IFT. This is because IFT takes long time to collect data (closed loop error and modeling error) and it takes same length time to process that data while MUC optimizes controller parameters directly.

The IFT investigated in this thesis is not suitable to control the speed of the DC Motor at constant speed since it operates on two different voltages at different given periods (in experiment1 operates at 3.0v and in experiment2 operates at 1.5v) making the speed not constant (at one period speed is high and at another period the speed is low).

On the other hand MUC proved to be more reliable and promising. Its results of simulation and those of the actual physical application were similar.

The output response, y converged within 8s confirming the claims of theory already highlighted in Chapter Three that MUC is more exact and faster as compared to IFT. It locks itself to one constant value after convergence is achieved rendering it to operate the DC Motor at one constant speed.

As stated above, the IFT and MUC are similar in many aspects. Both of them are model free, data-driven and gradient based. But there are some fundamental differences in their cost functions. For example IFT is viewed as a function of input and output data (u, and y) while MUC is viewed as a function of set point (r). Therefore MUC is doing optimization with one variable while IFT is doing it

with two variables. The process of IFT becomes run-to-run hence taking time which is not the case with MUC. Finally, IFT requires assumption not required by MUC.

IFT or MUC algorithm can be used in industrial applications with repetitive tasks such as robotic manipulators, disc-drive IC wafer production and steel-casting control to mention a few. It can also be effective speed controller for computer floppy drive motor, DVD player motor, VCR player motor, temperature controller and many other applications can be developed as already indicated in Chapter Two.

Now that it has been proved working on microcontroller, a very cheap and portable hardware, application on the mentioned systems can be made available.

7.2 Conclusion

Finally, this thesis investigated the feasibility of implementing the IFT and MUC algorithms on a microcontroller. The analysis and simulations carried out indicate that both are feasible to be implemented on a microcontroller. MUC converges faster than the IFT.

IFT can make a good automatic controller tuning system as opposed to be applied as a speed control of a DC Motor which is required to run at constant speed. For IFT to be robust (achieving convergence with zero modeling error) it requires features that prevent pole-zero cancellation to be incorporated in the code as highlighted previously.

MUC with its fast speed of convergence can be a good speed controller for a DC Motor as opposed to the use of IFT.

References

[1] X.-D.Li,J.K.L. Ho and T.W.S. Chow, “Iterative learning control for linear time –variant discrete systems based on 2-D system theory”, IEE Proceedings online no. 20041125, 2005.

[2] K J Astrom and B Wittenmark, “Adaptive Control”, Addison-Wesley, Massachusetts, 1989.

[3] Martin I.Machaba, “Investigations into Iterative Feedback Tuning”, Dept. Electrical Eng., Univ. Cape Town, 2004.

[4] Hakan Hjalmarsson, Michel Gevers, Svante Gunnarsson, Olivier Lequin “iterative feedback tuning: Theory and Applications”, Control Systems Magazine, v18, pp (26-41), 1998.

[5] K.Hamamoto, T. Fukuda and T. Sugic “Iterative Feedback Tuning of Controllers for two Mass-spring System with friction”, Control Engineering Practice, v11, PP (1061-1068), 2002.

[6] A. Karimi, L. MisKovic, D. Bonvin, “Iterative Feedback Tuning correction based controller tuning with application to magnetic suspension system”, Control Engineering Practice, v11, PP(1069-1078), 2002.

[7] J. Sjoberg, F.Debruyne, M. Agarwal, B.D.O Anderson, M. Gevers, F.J Kraus and N. Linard, “Iterative Controller Optimization for non linearn systems”, Control Engineering Practice, v11, PP(1079-1086), 2002.

[8] O. Lequin, M. Gevers, M. Mossberg, E. Bosmans and L.Triest, “Iterative Feedback Tuning of PID parameters: comparison with classical tuning rules”, Control Engineering Practice, v11, PP (1023-1033), 2003.

[9] F. De Bruyne, “Iterative Feedback Tuning for internal model controllers”, Control Engineering Practice, v11, PP (1043-1048), 2002.

[10] W.K. Ho, Y. Hong, A. Hansson, H. Hjalmarsson and J.W. Deng, “Relay auto-tuning of PID controllers using iterative feedback tuning”, Automatica, v31, PP(149-157), 2003.

[11] T. Meurers, S.M. Veres, and A.C.H. Tan, “Model-free frequency domain iterative active sound and vibration control”, Control Engineering Practice, v11, PP (1049-1059), 2002.

[12] Ari. G. Parttanen and Robert R. Bitmead, “The application of an iterative identification and controller design to a sugar cane crushing mill”, Automatic, v31, PP (1547-1563), 1995.

[13] H Hjalmarsson and T Birkeland, “iterative feedback tuning of linear time-invariant of MIMO systems,” 37th IEEE conference, PP (3893-3898), 1998.

[14] Joseph P. Gergen. Phil Hoang. Ephrem A, “Novel Digital Signal Processing Architecture with Micro-controller features”, Motorola inc., 6501 William Cannon, Texas 78735

[15] Martin Braae, “Advanced Topics in Control Engineering”, University of Cape Town, South Africa, 2004.

[16] R. Hildebrand, A. Lecchini, G. Solari, and M. Gevers, “Asymptotic accuracy of Iterative Feedback Tuning”, Center for Systems Engineering and Applied Mechanics, Univ Catholique de Louvain, 2004.

[17] Andrea Lecchini and Michel Gevers, “On Iterative Feedback Tuning for non-minimum Phase Plants”, Proc 41 IEEE Conf on Decision and Control, Las Vegas, USA, p4658-4663.

[18] David Calcutt, Fred Cowan, Hassen Parchizadeh, “Introduction to Microntrollers”, Microcontrollers an Application Based Introduction, Elsevier.

[19] B. W. Kernighan, D. M. Ritchie, “A Tutorial Introduction”, The C Programming Language, second edition, Prentice Hall Software Series, Englewood Cliffs, N. J. 07632.

[20] Schaule Stefan, “SPI Communication between DSP56F807C”, Motorola DSP Discussion Groups, 2003.

[21] Michael G. Safonov, “The Comparison of Unfalsified Control and Iterative Feedback Tuning”, Dept. of Electrical Engineering, University of Southern California, 2002.

[22] M. G. Safonov, and T.-C. Tsao, “The Unfalsified Control Concept and Learning”, IEEE Trans. On Auto. AC-42(6), pp 843-847, 1997.

[23] M. Jun and M. G. Safonov, “Controller Parameter Adaption Algorithm Using Unfalsified Control Theory and Gradient Method”, Proc., IFAC Congress, Barcelona, Spain, 2002.

[24] M. Jun and M. G. Safonov, “Automatic PID Tuning: An Application of Unfalsified Control”, In: proc. of the IEEE int. on CCA/CACSD. PP. 328-333. Honolulu, HI, 1999.

[25] Razavi and Kurfess, “Detection of Wheel and Work piece Contact/Release in Reciprocating Engine”, Int. Journal of Machine tools and Manufacture, v43 pp 185-191, 2003.

[26] Emmanuel Collins, “ Weigh Belt Feeder Adaptive PID Tuning”, int Journal of Adaptive Control, v15, pp 519-534, 1999.

[27] Kosut, “Semiconductor mfg. Process run-to-run tuning”, American Control Conference, v1, issue 4, ISBN 0-7803-3832-4, 1997

[28] Woodley, How & Kosut, “ECP Torsional disk control adaptive tuning”, ACC99

[29] S. M. Veres, and C. Graves, “Weak-duality in Worst-case Adaptive Control”, Proc. of American Control Conference, ACC. 94, Baltimore.

Appendix One

Iterative Feedback Tuning and Myopic Unfalsified Control Theory

This appendix chapter is mainly from [4, 15, and 23]. It is highly condensed only highlighting the main points.

A1. Iterative Feedback Tuning theory

The automated adjustment of parameters within control system to optimize a user defined cost function has been an interesting problem in control engineering for many decades. Iterative feedback theory is a method that claims to provide the design engineer with a viable tool to solve such problems [15] in that IFT adjusts the parameters of a feedback controller to ensure that the closed loop follows the desired model. This IFT concept is dealt with in detail in this appendix.

A1.1 The Dynamic Models

The equations governing the closed loop dynamics as per figure 3.1 ChapterThree (assuming the noise and disturbances are zero) are:

$$y = \frac{gk}{1+gk} r \qquad \text{A1.1}$$

Let T be the transfer function of the closed loop system and S be sensitivity function. These are defined as:

$$T = \frac{gk}{1+gk} \qquad (A1.2)$$

$$S = \frac{1}{1+gk} \qquad (A1.3)$$

Equation (A1.1) can then be written as:

$$y=Tr \qquad (A1.4)$$

Equations for desired model for open loop system yields:

$y_m = mr$ (A1.5)

$e_m = y - y_m$ (A1.6)

The performance of the controller, k is assumed to be quantified by the error, e_m that is clearly dependent on the parameter, ρ of the controller, k.

Where ρ is a parameter vector containing two components for the case of type 1 (one degree of freedom) controller given as?

$\rho = [\rho_0, \rho_1]$ (A1.7)

A1.3 The Cost Function

The cost function is a scalar based and is given by

$$J = \frac{1}{2}\sum_{1}^{N} e_m^2(t) \quad \text{(A1.8)}$$

The error, e_m, is normally collected first over a time period of N-length samples before summing and storing in the scalar function, J. J is what is known as the cost function and is used in the next experiment for parameter updating in case of first degree of freedom controller.

A1.4 Criterion Minimization or Gradient of the Cost Function

From equation (1.8) the cost function or quadratic criterion, J is defined. It is this function that is minimized through choosing optimal parameters (ρ_0 andρ_1) for a given controller. This controller (PI) is chosen because it is a very common control law as mentioned in Chapter Three and hence a reasonable stating point, or possibly the only one to consider because of its importance to industrial applications.

The model for a controller, k in figure A1.1 is given as

$$K(z) = \frac{\rho_1 z + \rho_0}{(z-1)} \quad \text{(A1.9)}$$

In equation (1.9), ρ_0 and ρ_1 are parameters of a chosen controller that need to be chosen to optimize the performance of the control system as defined by the quadratic criterion, J. These two parameters are elements in a vector of parameters, $\rho = [\rho_0 \ \rho_1]^T$.

To choose optimal parameters, ρ_0 andρ_1, the gradient of quadratic criterion, J is taken with respect to controller parameters. The gradient of criterion is represented by:

$$\frac{\partial J}{\partial \rho} = -\sum_1^N e_m * \frac{\partial e_m}{\partial \rho} \quad \text{(A1.10)}$$

The gradient of the error, e_m with respect to vector of controller parameters, ρ depends on the actual response y and not the desired response y_m. Thus the following equations provide the gradient for the cost function in terms of the elements of the controller parameter vector.

$$\frac{\partial e_m}{\partial \rho_0} = \frac{\partial y}{\partial \rho_0} \quad \text{(A1.11)}$$

$$\frac{\partial e_m}{\partial \rho_1} = \frac{\partial y}{\partial \rho_1} \quad \text{(A1.12)}$$

In order to determine the parameter that minimizes the quadratic criterion in equation (A1.10), one needs to find the derivative of $\frac{\partial e_m}{\partial \rho}$. This is equal to $\frac{\partial y}{\partial \rho}$, because the desired model, m(s) is a constant and hence independent of the controller parameters (ρ). The required gradient becomes:

$$\frac{\partial y}{\partial \rho} = \frac{1}{k} * \left(\frac{g * k * r}{1 + gk} \right) * \frac{\partial k}{\partial \rho} - \frac{1}{k} * \left(\frac{g^2 * k^2 * r}{(1 + gk)^2} \right) * \frac{\partial k}{\partial \rho} \quad \text{(A1.13)}$$

By substituting equations (A1.2) and (A1.3) into (A1.13) the derivative can be rewritten as:

$$\frac{\partial y}{\partial \rho} = \frac{1}{k} * \left[T * r - T^2 * r \right] \tag{A1.14}$$

Thus equation (A1.14) is a simplified version of (A1.13) and is the one that is used in IFT method to adjust or tune the controller parameters.

In the general two degrees of freedom IFT controller, three experiments are needed in the algorithm but this one-degree of freedom controller requiring two experiments to be generated. Thus only two N- set point signals are needed, $r_j^{(i)}$. Where i = 1,2. The corresponding output signal is represented by:

$$r_j^{(i)} = r,\ y^{(i)} = T*r \tag{A1.15}$$

$$r_j^{(2)} = (r_j^{(1)} - y^{(1)}),\ y^{(2)} = T*(r_j^{(1)} - y^{(1)}) = T*r - T^2*r \tag{A1.16}$$

Equation (A1.15) is generated in experiment 1 and equation (A1.16) is generated in experiment 2.

It can be observed that the signal $y^{(2)}$ generated in equation (A1.16) is exactly what is needed in equation (A1.14), therefore equation (A1.14) can be rewritten as:

$$\frac{\partial y}{\partial \rho} = \frac{1}{k} \frac{\partial k}{\partial \rho} y^{(2)} \tag{A1.17}$$

In this application the controller parameters form a vector and the update of controller parameters is given by the following equations:

$$\rho_{0_{j+1}} = \rho_0 - \gamma * r_{11} \frac{\partial J}{\partial \rho_0} - \gamma * r_{12} \frac{\partial J}{\rho_1} \tag{A1.18}$$

$$\rho_{1_{j+1}} = \rho_1 - \gamma * r_{21} \frac{\partial J}{\partial \rho_0} - \gamma * r_{22} \frac{\partial J}{\rho_1} \tag{A1.19}$$

These two equations (A1.19) and (A1.20) are derived from a general equation (A3.21) below.

$$\rho_{i+1} = \rho_j - \gamma_j R_j^{-1} \frac{\partial J}{\partial \rho}(\rho_j) \tag{A1.20}$$

Equation (A1.20) above can also be expressed in matrix form as shown in equation (A1.21)

$$\begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix} = \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix} - \begin{bmatrix} \gamma_0 0 \\ 0\gamma_1 \end{bmatrix} \begin{bmatrix} r_{11} r_{12} \\ r_{21} r_{22} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial \rho_0} \\ \frac{\partial J}{\partial \rho_1} \end{bmatrix} \tag{A1.21}$$

R can be the identity matrix or the Hessian of J:

$$\text{R} = \frac{1}{N} \sum_1^N \begin{bmatrix} \partial y / \partial \rho_0 \\ \partial y / \partial \rho_1 \end{bmatrix} \begin{bmatrix} \partial y / \partial_0 & \partial y / \partial \rho_1 \end{bmatrix} \tag{A1.22}$$

The constants r_{11}, r_{12}, r_{21}, and r_{22} are the elements of the matrix R_j given in equations (A1.19) and (A1.20).

The set of equations, (A1.19) and (A1.20) imply that the parameters will not be updated once the process response matches the model exactly.

A1.5 Myopic Unfalsfied Control Theory

Definination 1 : Unfalsification : A controller is said to be **falsified** by measurement information is sufficient to deduce that the performance specification $((r, y, u) \in T_{spec} \forall r \in R$ would be violated if that controller were in the feedback loop. Otherwise, the controller is said to be unfalsified.

Definition 2: Fictions Reference Signal $\tilde{r}$: Given measured data (u,y), a definition is given for each candidate controller k the fictitious reference $\tilde{r}$ (k) to be any signal with the property that if candidate controller K had been in the

control system during the period when the input -output data (u,y) was collected, and the signal $\tilde{r}$ (K) had been applied to the system, then measured data (u,y) would have been reproduced.

Because the data (u,y) may have been collected when one or more controllers other than k were in the feedback loop $\tilde{r}$ (k) is in general not the same as the actual reference signal *r.* this is why it is called a fictitious signal.

By computing the fictitious reference signal or signals associated with each candidate controller in real time, controller falsification and parameter tuning is done without inserting every candidate controller into the closed –loop system.

Based on the theorem in given in definition 1, a candidate controller k(θ) is unfalsified at time *t* by plant data u (τ), $(\tau \in [0,t]$ if and only if $\tilde{J}(\theta,\tau) \leq 0 \forall \tau \in [0,t)$

Where $\tilde{J}(\theta,\tau) = \nabla - \mathrm{p}(\tau) + \int_o^t \mathrm{T}_{spec}(r_i(\theta,\zeta)y(\zeta),u(\zeta),y(\zeta),(\zeta \in [0,t]$ are measured past plant data , and $\tilde{r}_i(\theta,\zeta)$ denotes the fictitious reference signal for the controller K (θ).

Since the objective of unfalsified control is to adjust the controller parameter vector θ so as to satisfy the given performance specification (as in Defifnition 1), one approach to achieving this objective is to adjust θ in the steepest decent direction $-\nabla\tilde{J}(\theta,t)$ so that the performance specification $\tilde{J}(\theta,t)$ tends to decrease whenever the currently active controller's parameter vector θ is falsified, viz.,

$\theta = \lambda\nabla\tilde{j}(\theta,t)$ Where $\gamma >$ 0 is a design constant that determines the rate of adaptation.

$\nabla \tilde{J}(\theta,t) \hat{=} [\frac{\partial \tilde{J}(\theta,t)}{\partial \theta_1} \quad \frac{\partial \tilde{J}(\theta,t)}{\partial \theta_2} ..... \frac{\partial \tilde{J}(\theta,t)}{\partial \theta_n}]^T = \int_0^T T_{spec(\tilde{r}_i}(\theta,\zeta),u(\zeta)).\nabla \tilde{r}(\theta,\zeta)d\zeta$ is the gradient of $\tilde{J}(\theta,t)$ with respect to θ, where

$$\nabla \tilde{r}(\theta,\zeta) = -K(\theta)K(\theta)^{-1}u(\zeta)$$

$$\nabla K(\theta) = [\frac{\partial K(\theta)}{\partial \theta_1} \quad \frac{\partial K(\theta)}{\partial \theta_2} \quad ..... \frac{\partial K(\theta)}{\partial \theta_n}]^T$$

Since u (*t) = (*θ)($\tilde{r}(t) - y(t)$

Therefore, controller parameter adaptation rule can be expressed as:

$$\theta = \begin{cases} \int_0^y \frac{\partial T_{spec}(\tilde{r},\zeta)}{\partial \tilde{r}} K(\theta)^{-1}\nabla K(\theta)K(\theta)^{-1}u(\zeta)d\zeta, & \text{If } \tilde{J}(\theta,\tau) > 0, \forall \tau \in [t_0,t] \\ 0, & \text{If } \tilde{J}(\theta,\tau) \le 0, \forall \tau \in [t_0,t] \end{cases}$$

Appendix Two

Introduction to Microprocessors and Microcontrollers

Microprocessors or Microcontrollers are widely used, as controlling component in all kinds of instruments. In this case the microcontroller with its peripheral extensions is the major responsible component for the functionality of any instrument, the reason being with the ease in which they are applied and manipulated [19].

Before embarking on implementation of IFT algorithm into microcontroller, it is vital to understand the major blocks of microcontrollers.
A microcontroller evolved from the microprocessor therefore it is imperative to talk about a microprocessor before microcontroller is discussed.

A microprocessor is a device containing functions equivalent to a small computer's central processing unit (CPU). It has

(1) control circuitry,
(2) an arithmetic logic unit (ALU),
(3) general purpose registers,
(4) address/program counter.

To execute a program, CPU sends out to memory the address of the location of the code for the first instruction to be executed. The CPU sends also a memory enable signal to fetch the instruction from addressed memory location that is decoded and executed. After each operation, the program counter is incremented to the address of the next instruction or data stored in memory a procedure known as fetch and execute.

A microcontroller is simply a complete computer system comprising at least three major components:

(1) the microprocessor (CPU),

(2) memory and

(3) input-output peripheral components

All these three elements are embedded in one single chip. A microcontroller could be a general-purpose computer or a system designed to fulfill a special task [19]. Figure AP2.1 shows a block diagram of a microcontroller to complete the definition of a microcontroller.

There are a number of other common characteristics that define microcontrollers [18]. If a computer matches a majority of these characteristics then it can be classified as a microcontroller.

Microcontrollers may be:

(1) Embedded inside some other device (often a consumer product) so that they can control the features or actions of the product. Another name for micro-controller is therefore an embedded controller.

(2) Dedicated to one task and run one specific program. The program is stored in ROM and generally does not change.

(3) A low-power device. A battery-operated micro-controller might consume as little as 50 mill watts.

A microcontroller may take an input from the device it is controlling and controls a device by sending signals to different components in the device [18].

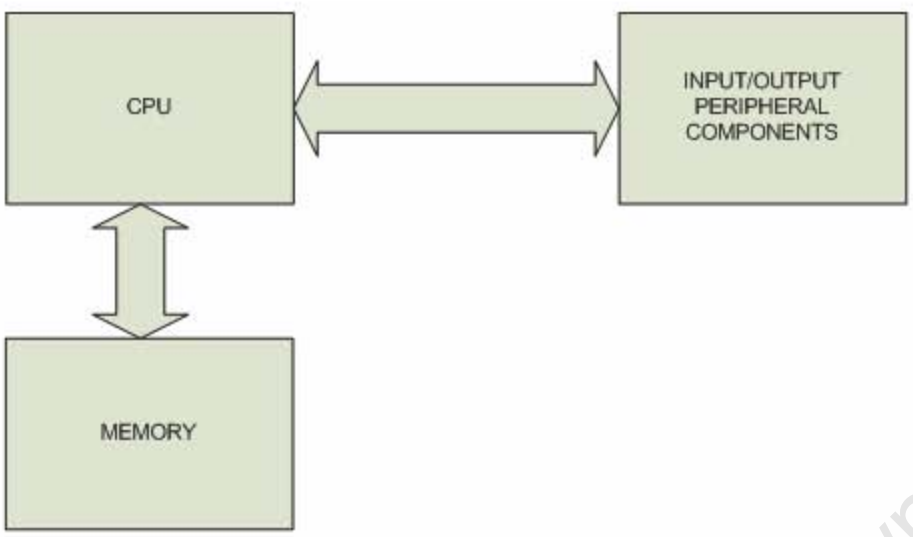

Figure AP2.1 Block diagram of a microcontroller

Appendix Three

IFT and MUC Algorithms Code

Ap3.1 IFT Algorithm Code

```
//: main.c
// {L} experiment1 and experiment2
// WRITTEN BY: Himunzowa Grayson
// FOR COURSE:
// PURPOSE:
// routine for IFT algorithm
//*************************************************
// for inclusion of cpu registers (automatically generated)
# include <ctype.h>
//*************************************************
//Automatically generated includes for shared modules
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
//*****************************************************
//declaration of function prototypes
#include “experiment1.h” // header file for experiment1 function
#include “experiment2.h” //  header file for experiment2 function
*****************************************************
Uword16 Sample; // ADC sample declared global
Double rt; // set point declared global
Double alf0, alf1; //controller parameters declared global
Double et1a[N] //closed loop error array declared global
Double emta[N] // modeling error array declared global
//*****************************************************
```

```
Int main (void)
 {
 alf0 = 0.3; alf1 = 0.3; // initializing parameters
 for(;;) // infinite loop
 {
 experiment1() // calls experiment1 function
 experiment2() // calls experiment2 function
 }
 return 0;
 }
```

```
// end of main body program
// : experiment.c
// {L} sig.h, ADC.h, and DACSpi.h
// WRITTEN BY: Himunzowa Grayson
// FOR COURSE:
// PURPOSE:
// routine for IFT algorithm experiment1 and experiment2
functions
**********************************************************
#include <stdio.h>
#include <math.h>
**********************************************************
#include "sig.h"
#include "adc.h"
#include <ctype.h>
#include "dacSpi.h"
**********************************************************
//Include shared modules, which are used for whole project
#include "PE_Types.h"
```

```
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
************************************************************
#ifndef EXPERIMENT_H
#define EXPERIMENT_H
// Prototype declaration
void experiment1();
void experiment2();
#endif; //EXPERIMENT_H
//***********************************************************
```

//Program for experiment1 function begins here

```
void experiment1()
{
// declaration and initialization
int j,n = N; //N is chosen by user with consideration of hardware
limitations
extern double rt; //Declaration of set point to experiment1
double am,p;
double uta; //Controller output signal
extern double alf0, alf1; //Controller parameters
extern UWord16  Sample; //ADC sampling
double yta; //output from process
double ymta; //output from the desired model
extern double et1a[N]; //Array for collection of closed loop error
extern double emta[N]; //Array for collection of modeling error
am = 1.0; p = 0.003;
uta = 0;
for(j = 0; j < n; j++)
        { //N-length loop
```

```
        signl(); //Call signal as a set point
        adc(); //Call ADC for sampling
        yta =(3.3*Sample)/1020; //Convert the digital value to analogue
        value
        et1a[j] = rt - yta; //Closed loop error
        uta = alf1*et1a[j] + alf0*et1a[j] + uta; //Controller output signal
        //yta = 0.5*yta + uta;//process plant only for simulation use
        ymta = am*rt - am*p*rt + p*ymta; //Desired model
        emta[j] = yta - ymta; //Outer or adaptive loop
        }// end of the for loop
        }// end of experiment1 function

void experiment2()
{
int j,n = N;
double dy_dalf0_tot;
double dy_dalf1_tot;
double dj_dalf0_tot;
double dj_dalf1_tot;
double gamaz;
double uta2 = 0;
extern double alf0, alf1;
extern UWord16  Sample;
double yta2;
double et2a;
double dy_dalf0t;
double dy_dalf1t;
extern double et1a[];
extern double emta[];
dy_dalf0t = 0.03; // Initialization of the gradient
```

```
dy_dalf1t = 0.03;
gamaz = 0.002;
dy_dalf0_tot = 0;
dy_dalf1_tot = 0;
for(j = 0; j < n; j++)
        {
        adc(); //Call ADC function
        yta2 = (3.3*Sample)/1020; //Convert digital value to analogue
        et2a = et1a[j] - yta2; //Calculate the error between the set point
        and the output
        uta2 = alf1*et2a + alf0*et2a + uta2; //calculate control signal,
        uta2 = uta2 + 1.65; //for biasing the controller
        yta2 = 0.5*yta2 + uta2; //Software Plant for use in simulations
        dy_dalf0t = (yta2 - alf0*dy_dalf0t)/alf1; //Calculate the output
        gradient
        //with respect to the controller parameter
        dy_dalf1t = (yta2 - alf0*dy_dalf1t)/alf1; //Calculate the output
        gradient
        //with respect to the controller parameter
        dy_dalf0_tot = dy_dalf0_tot + (emta[j]*dy_dalf0t); //Calculate the
        summation of the output
        //gradient
        dy_dalf1_tot = dy_dalf0_tot + (emta[j]*dy_dalf1t); //Calculate the
        summation of the output
        //gradient
        dj_dalf0_tot = dy_dalf0_tot/n;
        dj_dalf1_tot = dy_dalf1_tot/n;
        //updating controller parameter
        alf0 = alf0-((gamaz*dj_dalf0_tot) - (gamaz*dj_dalf1_tot))/n;
        alf1 = alf1-((gamaz*dj_dalf0_tot) - (gamaz*dj_dalf1_tot))/n;
```

```
        //spi_setup();
        }
    }
    ** : signal.c
    ** {L}
    ** WRITTEN BY: Himunzowa Grayson
    ** FOR COURSE: EEE 502 (MSc Eng. Thesis)
    ** PURPOSE:
    ** routine for generating set point signal for experiment1
    ******************************************************
#include "PE_Types.h"
    #include "PE_Error.h"
    #include "PE_Const.h"
    #include "IO_Map.h"
    ************************************************************
    #ifndef SIG_H
    #define SIG_H
    void signl();
    #endif; //SIG_H
    ************************************************************
    extern double rt;
    double rtmin;
    int count
    er = 0;
    int Anocounter = 0;
    void signl()
    {
    rt = 3.15; // maximum amplitude
    rtmin = 1.5; // minimum amplitude
    counter++;
```

```
        if(counter>20)
                {
                rt =rt - rtmin;
                Anocounter++;
                if(Anocounter>20)
                        {
                        counter = 0;
                        Anocounter = 0;
                        }
                }
        }
        //: experiment.c
        // {L}
        // WRITTEN BY: Himunzowa Grayson
        // FOR COURSE: EEE502 (MSc Eng. Thesis)
        // PURPOSE:
        // routine for ADC sampling
        ************************************************************
        #include "PESL.h"
        #include <stdio.h>
        ************************************************************
        #ifndef ADC_H
        #define ADC_H
        void adc();
        #endif //ADC_H
        ************************************************************
void adc()
        {
        unsigned int Information;
        extern UWord16  Sample;
```

```
//End of Processor Expert internal initialization.
PE_low_level_init();
/* Setup GPIO A0,A1,A2,A3,A4,A5,B0,B1,B2,B3 to control LEDs */
//PESL( GPIOA, GPIO_SETAS_GPIO, BIT_0 | BIT_1 | BIT_2 | BIT_3 |
BIT_4 | BIT_5);
PESL( GPIOB, GPIO_SETAS_GPIO, BIT_0 | BIT_1 | BIT_2 | BIT_3);
//PESL(GPIOA,GPIO_SETAS_OUTPUT, BIT_0 | BIT_1 | BIT_2 | BIT_3 |
BIT_4 | BIT_5);
PESL(GPIOB,GPIO_SETAS_OUTPUT, BIT_0 | BIT_1 | BIT_2 | BIT_3);
//PESL( GPIOA, GPIO_CLEAR_PIN, BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4
| BIT_5);
PESL( GPIOB, GPIO_CLEAR_PIN, BIT_0 | BIT_1 | BIT_2 | BIT_3);
/* Use loop sequential mode for sampling */
PESL( ADCA, ADC_SET_SCAN_MODE, ADC_SCAN_LOOP_SEQUENTIAL);
/* Connect Sample 0 with analog line 4 */
PESL( ADCA, ADC_SET_LIST_SAMPLE0, ADC_CH0);
/* Disable sampling of all Samples except for Sample 0 */
PESL( ADCA, ADC_WRITE_SAMPLE_DISABLE, 0x00FE);
/* Clear STOP bit in ADCR1 register */
periphBitClear(0x4000, &ArchIO.ADCA.ADCA_ADCR1_STR.Word);
/* Enable ADC sampling: set START bit in ADCR1 register */
PESL( ADCA, ADC_START, ...);
//while(1)
//{
        //UWord16  Sample;
        /* While Sample 0 is not ready */
        while(1)
        {
if (periphBitTest(0x0001,&ArchIO.ADCA.ADCA_ADSTAT_STR.Word) != 0)
                {
```

```
                    break;
                }
            }

            /* Read ADC Sample 0 *
            Sample = PESL( ADCA, ADC_READ_SAMPLE, 0);
            Sample = (Sample>>5) & 0x07FF+00;
            //printf("%d\n", Sample);
            //Information = Sample + 0x3100;
            //SM1_SendChar(Information);
             // Bit1_NegVal();
             // FC1_Enable();
             // FC1_Reset();
             // Bit1_NegVal();
            //break;
        //}
}
```

```
**  / / : dac.c
**  / / {L}
**  / / WRITTEN BY: Himunzowa Grayson
**  / / FOR COURSE: EEE502 (MSc Eng. Thesis)
**  / / PURPOSE:
**  / / routine for DAC interface
**  ***************************************************
#include "PE_SPI.h"
#include "PE_GPIO.h"
**************************************************************
#ifndef DACSPI_H
#define DACSPI_H
#include <stdio.h>
```

```
#define SPI_SS_DEACTIVATE GPIO_B_DDR |= 0x0080;
#define SPI_SS_ACTIVATE GPIO_B_DDR &= ~0x0080;
void spi_setup();
#endif // DACSPI_H
//*********************************************************
// the master setup:
void spi_setup() {
//periphBitSet(0x0070, &GPIOE_PER
    GPIO_E_PER |=0x0070; /* assign SCLK, MOSI & MISO (GPIOE4-6) to SPI
    peripheral*/
//periphBitChange( 0010, &GPIO_B_DR );
//GPIO_B_DR = ~0x0010; /* GPIOE7 will serve as /SS */
//GPIO_B_DDR |= 0x0080; /* it is output*/
//SPI_SS_DEACTIVATE; // SS must activate before write to SPDTR
// and deactive after for synchronisation !
    *(unsigned short*)(SPSCR) = 0x08C0; /* MSB first, Master mode,
    CPOL=0, CPHA=1, SPI enable */
*(unsigned short*)(SPDSR) = 0x000F; /* 12 bits */
//*(unsigned short*)(SPDTR) = Information;
}
```

Ap3.2 MUC Algorithm Code

```
// WRITTEN BY: Himunzowa Grayson
// FOR COURSE: EEE 502 (MSc Eng. Thesis)
// PURPOSE:
// routine for MUC Algorithm code
//*************************************************
#include <ADUc7024.H> // ADUc7024 MPU Register
#include <stdio.h> // For Used Function printf
#include <math.h> // ADUc7024 MPU Register
```

```
//*************************************************************
/* pototype  section */
void mainController(void);
void parameterUpdating(void);
void parameterUpdatingI(void);
float simpson (void);
void delay(unsigned long int);// Delay Time Function
int putchar(int ch);// Write Data to UART
int getchar(void);// Read Data From UART
void signal(void);
void adc (void);
int Dac (void);
//*************************************************************
float erro[500];
unsigned int val;
unsigned int val1;// ADC Result (HEX)
float simPson;
float rt;
float Theta;
int counter;
int Anocounter;
float intergral;
float u;
//*************************************************************
int main(void)
{
//float y ;
int counter;
int Anocounter;
Theta = 10.0;
```

```
intergral = 10.0;
//y = -3.0;
counter = 0;
Anocounter = 0;
while(1)
        {
        delay(40000000);
        mainController();
        //delay(40000000);
        }
}
//*********************************************************
void mainController()
{
        // declaration and initialization
        extern unsigned int val;// ADC Result (HEX)
        extern unsigned int val1;// ADC Result (HEX)
        int i;
        float y;
        float p;
        extern float erro[500];
        //float erro[100];
        float simPson;
        extern float rt;
        extern float intergral;
        float costFunction;
        float costFunctionI;
        float costfini;
        float yn2;
        extern float Theta;
```

```
extern float u;
int Counter;
Counter = 0;
p =0.01;
costfini = 0.0;
y = 0.0;
yn2 = 0.0;
for(i=0;i<500;i++)
{
Counter++;
signal();
adc ();
y = val * (2.50 / 4096.0); // Volt = ADC Result x [2.5V / 4095]
erro[i] = rt - y;
//u = 1*Theta*erro + 1*intergral*erro + u;
//u = 1*Theta*erro[i] + u;
if(i > 0)
        {
        u = Theta*erro[i] + intergral*0.1*erro[i-1] -Theta*erro[i-
1]+ u; // new controller
        //u = (Theta + intergral)*erro[i]- Theta*erro[i-1]+ u;
        }
        //y = 0.181269*y - 0.818731*yn2 + 0.090635*u;
        //yn2 = y;
        //y = 0.917404*y + 1.431382*u;// light disc model
        y = 0.97531*y + 4.2490*u; //Heavy disc model
//y = 0.917404*y + 0.431382*u;// light disc model modified
delay(5000);
//printf("  %1.1f\n",costFunction);// Display 3-Digit Result(0-2.5V)
delay(5000);
```

```
//printf("  %1.1f\n",y);// Display 3-Digit Result(0-2.5V)
//u = u*p;
//y = 0.5*y + u; //process plant only for simulation
//y = 0.9872462*y + 2.65*u;//process plant only for simulation use
val = u * (4096.0/2.50);
Dac ();
Theta = Theta + 1.0;
Theta = pow(Theta,2);
simpson ();
Theta = 1/(2*Theta);
intergral = intergral + 1.0;
intergral = pow(intergral,2);
simpson ();
intergral = 1/(2*intergral);
costFunction = -costfini + Theta*simPson;
if(costFunction < -3.0)
{
costFunction = -3.0; //soft limit
costFunction = costFunction * -1.0;
}
//if(costFunction < 0.0)
costFunctionI = -costfini + intergral*simPson;
if(costFunctionI < -3.0)
{
costFunctionI = -3.0; //soft limit
costFunctionI = costFunction * -1.0;
}
val1 = costFunction * (4096.0/2.50);
Dac ();
//u = pow(u,2);
```

```
        if(costFunction > 0.0)
        parameterUpdating();
        if(costFunctionI > 0.0)
        {
        parameterUpdatingI();
        }
        }// end of the for loop
}// end of mainController function
//*************************************************************
//declare variables
float simpson(void)
        {
        extern float simPson;
        float sum;
        int coefficient;
        float u; // x-axis points for simpson
        float factor; //simpson factor
        float change;// delta x
        float firsTerm; // variable to hold first term of simpson
        float Error; //Error bound for simpson
        float M;
        float lasTerm;
        float nextT;
        int uEnd, uBeg;
        int x;
        int h;
        int i;
        int noInterval;
        noInterval = 100;
        uBeg = 0;
```

```
uEnd = 2;//sampling frequency is 774ksps (1.29micro seconds)
u=1.0;
sum=0.0;
//double* nextTerm= new double[noInterval];
x = uEnd - uBeg;
h = pow(x,5);
factor=x/(3*noInterval);
M=30.0024*pow(noInterval,4)/h;
Error=M*h/(180*pow(noInterval,4));
change = x/noInterval;
u = pow(u,2);

for( i = 0; i <= noInterval - 1; i++)
        {
        u = u + change;
        nextT = pow(u,2);
//function to get coefficient of simpson term
        if( i==0||i%2==0)
        {
        coefficient=4;
        }
        else
        coefficient=2;
sum = sum + coefficient*nextT;
}
lasTerm = pow(u,2);
sum = lasTerm + firsTerm + sum;
Error = Error * factor * sum;
simPson = (factor * sum) - Error;
return simPson;
```

```
        }
//*********************************************************
void parameterUpdating()
{
float costfini;
extern float Theta;
extern float u;
int stepSize;
stepSize = 5;
costfini = 0.0;
Theta = pow(Theta,3);
Theta = (stepSize/Theta);
Theta = Theta * simPson;
}
//*********************************************
void parameterUpdatingI()
{
float costfini;
extern float intergral;
extern float u;
int stepSize;
stepSize = 5;
costfini = 0.0;
intergral = pow(intergral,3);
intergral = (stepSize/intergral);
intergral = intergral * simPson;
}

//*********************************************
```

```
void adc ()
{
        extern unsigned int val;// ADC Result (HEX)
        extern unsigned int val1;// ADC Result (HEX)
        int adc_scan;// ADC Channel Scan
        int i;
        GP1CON &= 0xFFFFFFCC;// Reset P1.1 & P1.0 Pin Function
        GP1CON |= 0x00000011;// Setup P1.1 = TXD & P1.0 = RXD

        // Initial UART = 9600BPS
        COMCON0 = 0x80;  // Setting DLAB
        COMDIV0 = 0x88;   // Setting DIV0 and DIV1 to DL calculated
        COMDIV1 = 0x00;
        COMCON0 = 0x07;  // Clearing DLAB
        // Power-ON ADC
        ADCCON  = 0x00000000;// Reset ADC Config
        ADCCON |= 0x00000020;// Power-ON ADC Function
        delay(1000);// Wait ADC Power-on Ready
        ADCCON |= 0x00001400;// ADC Clock = fADC/32
        ADCCON |= 0x00000300;// Acquisition Time = 16 Cycle Clock
        ADCCON &= 0xFFFFFFE7;// ADC = Single-End Mode
        ADCCON |= 0x00000004;// Continue Software Convert
        REFCON  = 0x00000001;// Used Internal 2.5V Reference
        ADCCON |=  0x00000080; // ADC Start Conversion
        for(i=0; i<2; i++)
        {
        adc_scan=0;
ADCCP = adc_scan;// Select Channel to Conversion
        delay(1000);// Wait Select Channel Ready
        while (!ADCSTA){};// Wait ADC Conversion Complete (Bit0="1")
```

```
        val = (ADCDAT >> 16)& 0x00000FFF;// Shift ADC Result to Integer

        delay(1000);
        }

    }
        int Dac (void)
        {
        extern unsigned int val;
        extern unsigned int val1;// ADC Result (HEX)
        //val = 0x0FFF;
        // Initial DAC0
        DAC0CON &= 0xDF; //DAC0 Used Sysytem Clock
        DAC0CON |= 0x10; // Enable DAC0
        DAC0CON |= 0x03; // DAC0 Output Range = AVDD..AGND
        // Initial DAC1
        DAC1CON &= 0xDF; // DAC0 Used Sysytem Clock
        DAC1CON |= 0x10; // Enable DAC0
        DAC1CON |= 0x02; // DAC0 Output Range = +Vref..AGND
        REFCON  = 0x01;  // Used Internal 2.5V Reference
        // val = 0x0000;
        DAC0DAT = (val << 16); // Update DAC0 Sine Output(0..3V3)
        DAC1DAT = (val1 << 16);
        return val;
        }
      //end of MUC program

//*************************************************************
    void signal()
        {
```

```
        extern float rt;
        float rtmin;
        extern int counter;
        extern int Anocounter;
        rt = 1.5; // maximum amplitude
        rtmin = 1.5; // minimum amplitude
        counter++;
                if(counter>100)
                        {
                        rt =rt + rtmin;
                        Anocounter++;
                        if(Anocounter>100)
                                {
                                counter = 0;
                                Anocounter = 0;
                                }
                        }
        }

/* Delay Time Function */
void delay(unsigned long int count1)
{
while(count1 > 0) {count1--;}      // Loop Decrease Counter
}

/***************************/
/* Write Character To UART */
/***************************/
int putchar(int ch)  // Write character to Serial Port
{
```

```
if (ch == '\n')
{
while(!(0x40==(COMSTA0 & 0x40)))// Wait TX Complete
{
        }
        COMTX = 0x0D;// Write CR
}
while(!(0x40==(COMSTA0 & 0x40)))// Wait TX Complete
{
}
return (COMTX = ch);
}

/* Read Character From UART */
/*****************************/
int getchar (void)     // Read character from Serial Port
{
while(!(0x01==(COMSTA0 & 0x01)))// Wait Receive Data Ready
{
}
return (COMRX);
}
//*************************************************************
```

Appendix Four

Visual Basic IFT Algorithm Code

Table AP4.1 Overview of the code Procedure 'Experiment1'

COMPUTER CODE	DESCRIPTION
alf0 = alf0z.Text	Read the initial parameters
alf1 = alf1z.Text	Read the initial parameters
yta(j) = sgGenerator(rGen, Plot_t)	Read the set point r, which is in vector format
yta(j) = yt	Store the actual response $y^{(1)}$
et1a(j) = rta(j) – yta(j)	Store the error, which is the set point to experiment 2
uta(j) = alf1 * et1a(j) + alf0 * et1a(j) + uta(j – 1)	Calculate the input signal, $u^{(1)}$
Ymta(j) = am * rta(j) – am * p * rta(j) + rta(j) + p * ymta(j)	Calculate the desired response, y_m
Emta(j) = yta(j) – ymta(j)	Calculate the error between the actual and the desired response
Ja = ja + 0.5 * emta(j)^2	Do the summation of the criterion J

Table AP4.2 Overview of the code Procedure 'Experiment 2'

COMPUTER CODE	DESCRIPTION
rt = et1a(j)	Read the set point, which was calculated and stored from experiment 1
yta2(j) = yt	Read the actual response for $y^{(2)}$
et2a(j) = rt – yt	Calculate the error between the set point and the output
uta2(j) = alf1 * et2a(j) + alf0 * et2a(j) + uta2(j-1)	Calculate the input signal, $u^{(2)}$
Ut = uta2(j)	Store the input
dy_dalf0t(j) = (yta2(j) – alf0 * dy_dalf0t(j))/ alf1	Calculate the output gradient with respect to the controller parameter ρ_1
dy_dalf1t(j) = (yta2(j) – alf0 * dy_dalf1t(j))/ alf1	Calculate the output gradient with respect to the controller parameter ρ_0
du_dalf0t(j) = (uta2(j) – alf0 * du_dalf0t(j))/ alf1	Calculate the input gradient with respect to the controller parameter ρ_1
du_dalf1t(j) = (uta2(j) – alf0 * du_dalf0t(j))/ alf1	Calculate the input gradient with respect to the controller parameter ρ_0
dy_alf0_tot = dy_alf0_tot + dy_dalf0t(j)	Calculate the summation of the output gradient with respect to the controller parameter ρ_0
dy_alf1_tot = dy_alf1_tot + dy_dalf1t(j)	Calculate the summation of the output gradient with respect to the controller parameter ρ_1
du_alf0_tot = du_alf0_tot + du_dalf0t(j)	Calculate the summation of the input gradient with respect to the parameter ρ_0

du_alf1_tot = du_alf1_tot + du_dalf1t(j)	Calculate the summation of the input gradient with respect to the parameter ρ_1

dj_dalf0 = dj_dalf0 – emta(j) * dy_dalf0t(j)	Calculate the criterion minimization for controller parameter ρ_0
dj_dalf1 = dj_dalf1 – emta(j) * dy_dalf1t(j)	Calculate the criterion minimization for controller parameter ρ_1
r11t = ((dy_alf0_tot)^2 + lamda * (du_alf0_tot)^2)/1000	Elements of matrix R_j
r12t = (dy_alf0_tot * dy_alf1_tot + (du_alf0_tot * du_alf1_tot))/1000	Elements of matrix R_j
r21t = (dy_alf1_tot * dy_alf0_tot + (du_alf1_tot * du_alf0_tot))/1000	Elements of matrix R_j
r11t = ((dy_alf1_tot)^2 + lamda * (du_alf1_tot)^2)/1000	Elements of matrix R_j
Alf0 = alf0 – (gamaz * r11t * dj_dalf0)/1000 – gamaz * r12t * (dj_alf1)/1000	The calculation of updating controller parameter ρ_0
Alf1 = alf1 – (gamaz * r21t * dj_dalf0)/1000 – gamaz * r22t * (dj_alf1)/1000	The calculation of updating controller parameter ρ_1

Appendix Five

Graphs for the Step Responses of the DC Motor

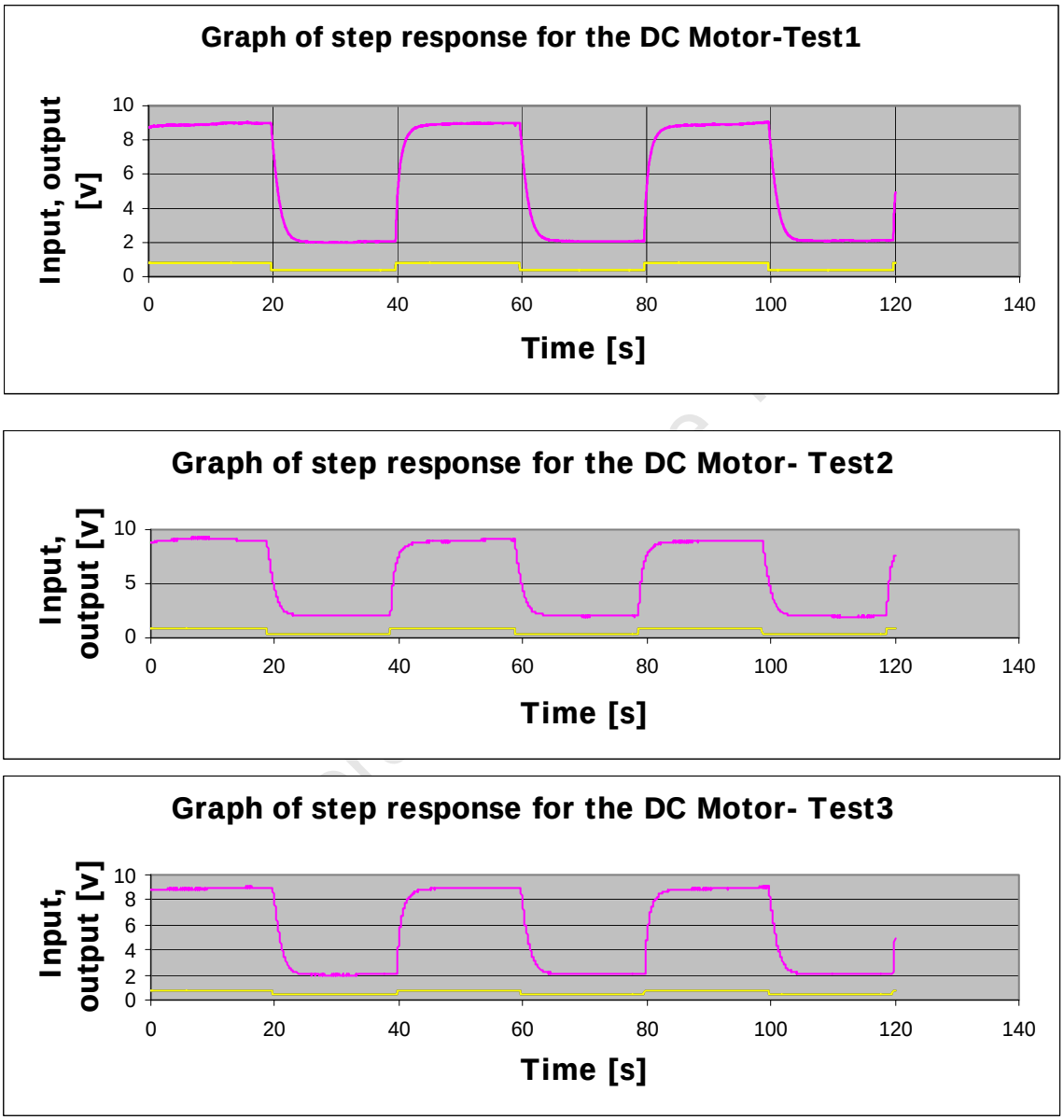

Figure AP5.1 Step response for the DC Motor light disc model.

Appendix Six

Graphs of input signals to the DC Motor

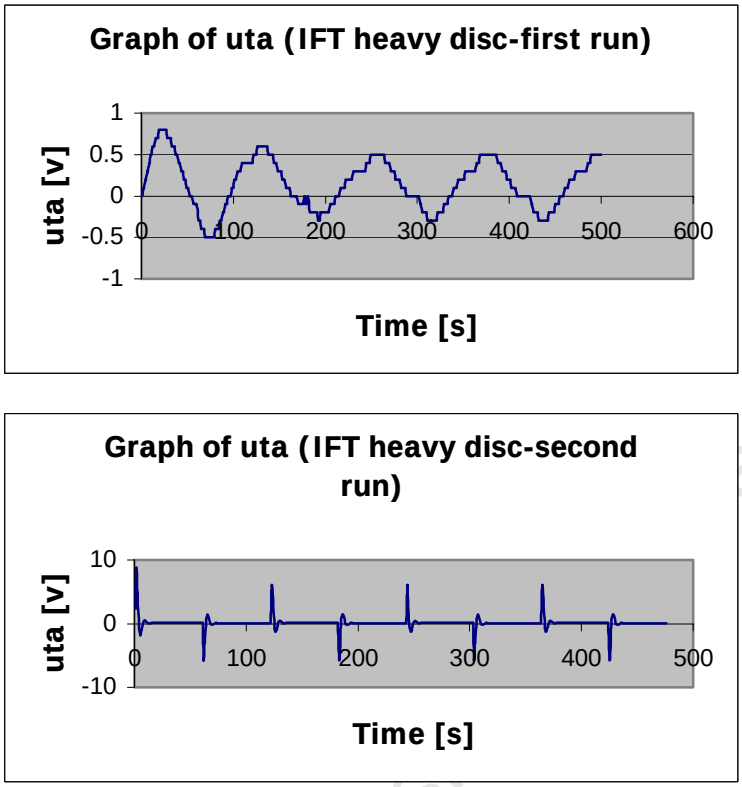

Figure AP6.1 Input signal (uta) for heavy disc model

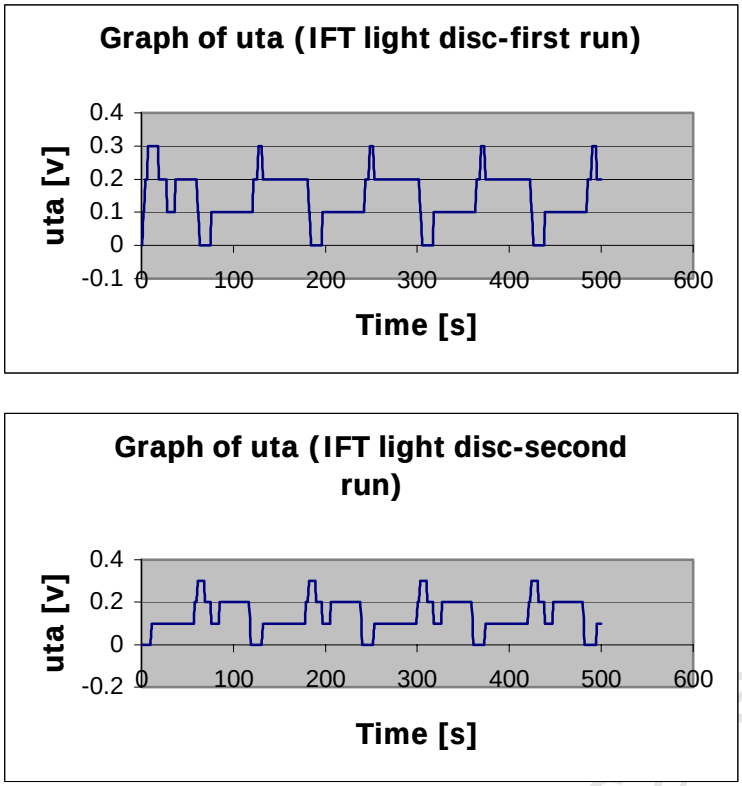

Figure AP6.2 Input signal (uta) for light disc models

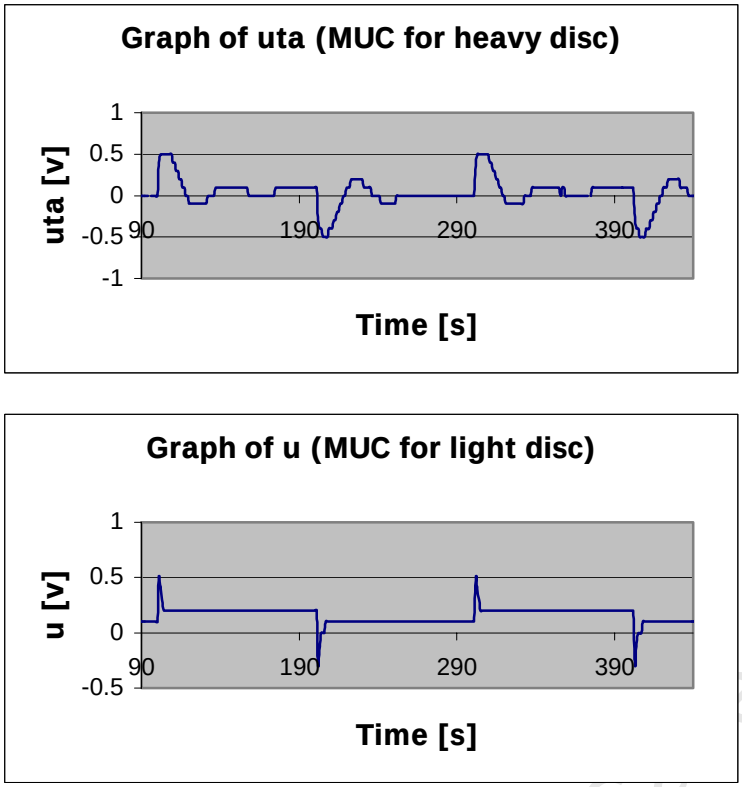

Figure AP6.3 Input signal (u) for heavy and light disc models